Sistemas informáticos

.Ingeniería Superior de Informática

Facultad de Informática, Universidad Complutense de Madrid

Memoria

.

Editor de terreno y máquina de estados para el desarrollo de videojuegos

Terrain and Finite State Machine tools for video game development

Autor:	*Alejandro Simón*
DNI:	*53386828-X*
Fecha:	Junio *2010*
Profesor director:	*Pedro Antonio González Calero*

El autor de este proyecto, Alejandro Simón Pérez alumno de la Facultad de Informática, autoriza a la Universidad Complutense de Madrid a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor, tanto la propia memoria, como el código, la documentación y/o el prototipo desarrollado.

Alejandro Simón

ÍNDICE

0. Resumen (español)

Este proyecto ha sido concebido para resolver dos de los problemas más importantes en el desarrollo de videojuegos. Ambos problemas serán tratados durante todo este documento, proponiendo diversas soluciones y explicando el por qué de la solución escogida en cada caso. Como producto final se han creado dos herramientas que pueden ser usadas para mitigar dichos problemas. Para demostrar su correcto funcionamiento y la consecución del objetivo marcado se incluyen pruebas y demostraciones de ambas herramientas.

Por un lado, la mayor parte de los videojuegos actuales requieren escenarios y terrenos realistas, que además puedan ser modificados dinámicamente durante el desarrollo de la partida. Esto dotará de un gran realismo a cualquier juego. En este ámbito se han investigado diferentes técnicas, las cuales serán explicadas en esta memoria. La herramienta desarrollada ha tenido como requisito indispensable disponer de un manejo fácil e intuitivo, ya que teóricamente está ideada para ser usada, entre otros, por diseñadores con el fin de crear distintos escenarios dentro de un juego. El nombre utilizado a lo largo de esta memoria para referirse a esta herramienta es Editor de Terreno o *Terrain Editor*.

El segundo problema abordado con el desarrollo de este proyecto es el de la animación y movimiento de los personajes en el juego. Desde el punto de vista del usuario se puede llegar a pensar que esta parte del desarrollo de un videojuego consiste únicamente en modelar y animar a los personajes en un software de modelaje 3D como puede ser *Maya*, *3D Studio Max* o *Softimage ModTool*, éste último ha sido el empleado para la realización del proyecto. Pero se debe tener en cuenta que además crear el modelo y las animaciones, se debe integrar este trabajo dentro del juego. Esto no es algo trivial y consume gran cantidad de tiempo y recursos. Sin lugar a dudas este proceso es uno de los más costosos y supone un cuello de botella en la planificación y en el desarrollo de cualquier videojuego. Es aquí donde la segunda herramienta desarrollada en este proyecto entra en escena. La aplicación, denominada *Finite State Machine tool*, supone un puente entre el trabajo de modelado y animación y el código del juego. Además sirve de ayuda directa a los diseñadores, ya que les permite visualizar sus creaciones tal como serán en el juego final. Utilizar esta herramienta en las primeras etapas del desarrollo puede ahorrar mucho tiempo y puede suponer un aumento en la calidad final del producto. Es importante aclarar en este punto que esta herramienta no son los modelos 3D o las animaciones. El contenido variará dependiendo del juego en el que la aplicación vaya a ser utilizado.

Ambas herramientas han sido desarrolladas utilizando *Microsoft XNA Framework* en el lenguaje *C#*, para dos plataformas diferentes: *Windows* y *Microsoft Xbox 360*. *XNA* es un f*ramework* de libre distribución que permite la utilización de un motor gráfico y físico junto con librerías específicas para el desarrollo de videojuegos tanto

para PC como para Xbox 360, Zune y, en la última versión del producto, Windows Mobile. Ésta se ha convertido en un herramienta muy popular entre el mundo del desarrollo de juegos independientes, por su potencia y accesibilidad. Las dos herramientas desarrolladas en este proyecto encajan perfectamente en este marco. Cambiando el contenido del material usado por ellas, como los modelos, texturas o animaciones, ambas herramientas pueden ser usadas para el desarrollo de prácticamente cualquier tipo videojuego que se quiere implementar con este *framework*. Sin embargo, se escogió un concreto juego en desarrollo. Gracias a ello es posible mostrar el potencial de las herramientas en un entorno real, con contenido propio de un juego. Los modelos 3D, animaciones y texturas utilizados por las herramientas provienen de este proyecto. Además de este juego, llamado Darksoul Paradox, ambas herramientas han sido utilizadas en otro juego como demostración. El lector podrá encontrar información detallada de todo ello en la sección 3.1: Origen y en la sección 7: Demostración.

El proyecto ha sido desarrollado en *Middlesex University* (Londres) y tuteado por el profesor director del proyecto, Pedro Antonio González Calero, desde la Universidad Complutense de Madrid. Por esta razón el grueso de la memoria del proyecto se encuentra escrito en inglés, así como los comentarios dentro del código, la interfaz de las herramientas y otros elementos.

Debido a que el proyecto consta de dos herramientas, cada sección de este documento será desarrollada para cada una. En los primeros puntos el lector encontrará una introducción y una descripción general del proyecto, seguido de una explicación detallada de las técnicas utilizadas más importantes. En la siguiente sección se incluye información y documentos (diagramas y gráficos UML) relevantes para el entendimiento del análisis y diseño seguido para la consecución del proyecto durante todo el desarrollo. A continuación se amplía la información relativa a las técnicas utilizadas completando la descripción presentada al comienzo de la memoria con los detalles relativos a la implementación de cada técnica. En este punto se incluyen partes de código relevantes, así como explicaciones. Para completar este trabajo escrito, se incluyen pruebas realizadas para demostrar la validez de las herramientas, como capturas de pantalla mostrando las distintas configuraciones y sus ventajas e inconvenientes. Además, se muestra como la herramienta *Terrain Editor* ha sido utilizada exitosamente en un segundo juego para PC y Xbox360, que aprovecha y saca partida a todo su potencial. Por último, pero no menos importante, en el último apartado se deja espacio para la auto-evaluación, la reflexión sobre el proceso seguido en cada fase y espacio para comentar futuras posibles ampliaciones y mejoras que harían ambas herramientas más potentes. Además, junto con esta memoria se incluyen tres apéndices: la bibliografía, una lista de los distintos recursos utilizados durante el desarrollo y una guía para la correcta instalación y manejo de las herramientas, incluyendo todos los controles disponibles, tanto como para el uso de teclado como de mando USB compatible con Xbox.

Técnicas utilizadas

Para cada herramienta se han elegido las técnicas óptimas que permiten conseguir el mejor rendimiento con la mejor calidad. Todas las técnicas seleccionadas son usadas en títulos de la industria actuales por lo que su buen resultado aplicado a un videojuego está prácticamente asegurado. A continuación el lector podrá encontrar un breve resumen de cada una de las técnicas, clasificadas según a la herramienta en la que fueron implementadas. Empezaremos con las técnicas utilizadas en el Editor de Terreno.

Mapa de alturas (Height mapping): esta técnica consiste en crear una malla de 3 dimensiones, que será un escenario, a partir de un mapa de alturas. Un mapa de alturas es una imagen en blanco y negro que representa las diferencias de altura en un terreno o una superficie. De este modo el color negro corresponde a la altura más baja, mientras que el blanco supone los puntos más altos del terreno. Las alturas entre la más alta y la más baja, tendrán el color entre el blanco y el negro correspondiente. Ésta es una técnica muy potente para crear escenarios de 3 dimensiones. Éste es un proceso muy popular y ha sido utilizado en juegos muy exitosos del mercado. Además de esto, en la herramienta implementada para esta proyecto, esta técnica también nos servirá para permitir modificaciones en el terreno dinámicamente, es decir mientras el juego trascurre. Modificando el mapa de alturas adecuadamente es posible alterar el terreno a nuestro antojo, y así de este modo simular aquellas acciones que puedan modificar el terreno, como explosiones. En el apartado 3.2.1 de esta memoria se incluye un ejemplo de un mapa de alturas y títulos reales que han utilizado esta técnica para recrear escenarios.

Mapa de normales (Normal mapping): es una técnica que se emplea para conseguir que el usuario tenga la impresión de que está viendo una superficie muy detallada cuando en realidad es una textura plana. Por tanto, esta técnica permite usar objetos con pocos polígonos pero consiguiendo un nivel de detalle alto, ahorrando así muchos cálculos de computación costosos. Esta técnica es perfecta para aplicar a un terreno en el que se quiere conseguir un resultado realista sin sobrecargar las capacidades de la CPU y GPU. Esto se consigue codificando el nivel de detalle de la superficie plana en su textura, en lugar de usar una superficie de 3 dimensiones muy detallada, es decir con muchos polígonos, y, por tanto, muy compleja de calcular. Así esta técnica consiste en almacenar la normal de cada pixel como un vector de 3 dimensiones dónde las componentes son los valores RGB. Después estas normales serán usadas para calcular la luz que incide y es reflejada en ese pixel. Las normales son guardadas en un “espacio de coordenadas de tangentes” (tangent space coordinate system) para poder ser utilizadas. Para estos cálculos se necesitan dos texturas: por un lado la textura de la superficie deseada (una roca, tierra, vegetación, metal…). Por otro lado, un mapa de normales de esa misma textura. Esta imagen tiene un tono azulado y puede ser creado fácilmente usando un programa de edición gráfica como es Photoshop.

Iluminación - modelo Phong: éste es un modelo utilizado para calcular la iluminación local. El modelo describe como refleja la luz una superficie en un determinado punto a partir de una combinación de distintos colores: ambiente, difuso y especular. Esta técnica supone una manera eficiente de calcular la luz con un alto grado de detalle. Por ello el modelo Phong se utiliza ampliamente en la implementación de la iluminación en videojuegos. Este modelo ofrece una ecuación para resolver la iluminación en cada punto. Los detalles de esta ecuación y sus consecuencias son estudiados en detalle en la sección 3.2.3 de este mismo documento.

Uso de texturas basado en la altitud del terreno: una vez creada la malla 3D que será el terreno en el juego, sobre ésta habrá que aplicar diferentes texturas para conseguir escenarios más reales. Esta técnica se encarga de ello. Se aplican diferentes texturas dependiendo de la altitud del terreno en cada punto. Además para los valores frontera se mezclan dos texturas diferentes para suavizar los cambios bruscos. Este método puede ser muy útil y potente si se eligen texturas y altitudes adecuadas.

Sistema de detección de colisiones con el mapa de alturas: para que el terreno sea totalmente utilizable, debe disponer de un sistema que detecte cuando se produce una colisión entre un objeto del entorno (como puede ser uno de los personajes) con el propio terreno. Una vez que se detecte dicha colisión, será el objeto el que actúe consecuentemente. En el caso de un modelo, la acción será no traspasar nunca el terreno manteniéndose justamente sobre él. Si por el contrario fuera un proyectil el que chocara contra el terreno entonces ambos elementos reaccionarían al impacto. El proyectil desapareciendo y creando una explosión, mientras que el terreno se encargaría de “deformarse” produciendo un cráter acorde con dicha explosión.

Máquinas de estados y animación “keyframed”: una máquina de estados es un modelo de comportamiento formados por estados, acciones y transiciones entre estados. En nuestro caso, los estados serán posibles comportamientos de cada personaje. Las acciones vendrán determinadas en su mayoría por entrada del software, es decir, el teclado o el mando. Estás acciones (pulsar un botón, la palanca de dirección), originarán cambios de estado. Las máquinas de estados se usan con frecuencia en el mundo de los videojuegos en las animaciones y la inteligencia artificial por sus múltiples ventajas: son simples, flexibles y fáciles de mantener. Además permiten la implementación de comportamientos complejos mediante la encapsulación de máquinas de estados más simples. Así mediante un jerarquía es posible crear tantos comportamientos y tan complejos como sean necesarios. Para implementar una máquina de estados el primer paso es diseñar un diagrama incluyendo todos los estados, transiciones y acciones pertinentes. Una vez que este diagrama está terminado, revisado y cumple con todos los requisitos necesarios se puede implementar y codificar. Todos los diagramas diseñados para este proyecto pueden ser consultados en la sección 5.2, correspondiente al diseño de la máquina de estados.

En cuanto a la animación existen dos técnicas principales: animación “keyframed” y “skeletal animation”. La primera técnica permite crear una animación basada en puntos clave (keyframes) de una secuencia. Esta es la técnica elegida para implementar la locomoción y movimiento de los personajes en la herramienta desarrollada. Para implementar esta técnica se exporta, desde el programa de modelado al proyecto XNA, una malla estática por cada punto clave de la secuencia deseado. Una vez que todas las mallas han sido guardadas, la animación completa se obtendrá interpolando cada par de puntos clave. Interpolación es un método matemático que consiste en obtener nueva información (animaciones interpoladas) dentro de un determinado rango discreto de un conjunto de puntos de información (puntos clave). El resultado es una animación completa. El ojo humano es incapaz de distinguir la diferencia con una animación, digamos, real. Esto será así siempre que la animación disponga de la suficiente información, esto es que se mantenga en una cantidad de imágenes por segundo (*fps)* aceptable: del orden de entre 20 y 30. Todos los modelos usados en el desarrollo de este proyecto han sido implementados usando *Autodesk Softimage ModTool.* Ésta es una aplicación de modelaje y animación 3D utilizada en importantes títulos de la industria del videojuego como son: Half Life 2, Resident Evil o Metal Gear Solid. Una razón por la que se escogió esta técnica para la animación de personajes en la herramienta es porque es un proceso muy rápido, ya que todos los cálculos de interpolación son hechos directamente a nivel programación. Por otra parte, una desventaja de este método es que consumo gran cantidad de espacio, ya que es necesario guardar en memoria una malla por cada punto clave de la animación. Sin embargo, éste supone un precio muy bajo a pagar, en comparación de las grandes ventajas que supone.

Conviene explicar el segundo método, así el lector podrá tener una idea más clara del por qué de la elección de uno de ellos. Con “*skeletal animation”* cada personaje vertebrado es un conjunto de huesos (rig). Cada hueso dispone de una matriz de transformación: posición, escala y orientación. También tiene información del hueso que le gobierna a él y al resto de la jerarquía de huesos. Éste será su hueso padre (*parent bone*). Por tanto, la transformada completa de nodo hijo será el producto de su transformada y la de su padre. Así, las transformadas de toda la jerarquía se calculan dependiendo de los huesos principales. Mientras el personaje está siendo animado, cada transformada vinculada a cada uno de los huesos es recalculada. En este punto el lector podrá hacerse una idea de lo costoso que es este proceso. La ventaja es que se tiene prácticamente un control total sobre la animación pudiendo aplicar diferentes fuerzas sobre cada transformada para conseguir resultados más realistas como rozamientos, viento o simular distintos tipos de superficies.

Diseño

En esta sección nuestro lector podrá encontrar los diagramas de las máquinas de estados y los diagramas UML de cada una de las dos herramientas. El objetivo es mostrar como las herramientas fueron diseñadas y pensadas para cumplir ciertos objetivos primordiales. El principal requisito que ambas herramientas deben cumplir es que deben ser fácilmente integrables en otros proyectos. Esto se consigue implementando cada aplicación como un componente (Drawable Game *Component*). Así, por ejemplo, el Editor de Terreno está divido en tres componentes diferentes, el motor: en el que se incluyen las clases propias del control de la entrada y la cámara, el menú: correspondiente al control y gestión de los diferentes menús de la aplicación y el componente del terreno propiamente dicho: dónde se incluye toda la funcionalidad propia de la creación y modificación de éste. Así, para integrar esta herramienta, no hay ningún tipo de dependencia entre las clases que pertenecen puramente al terreno con las otras clases que dependerán de cada proyecto. Por otra parte, ambas herramientas deben garantizar que el contenido sea fácilmente ampliable o sustituible por el contenido propio del proyecto en el que será utilizada. Esto es especialmente importante en la Máquina de Estados. Sea cuidado mucho el diseño de ésta, para que agregar nuevos personajes, modificar los existentes o añadir nuevos estados sea lo más sencillo posible.

Además del diseño y análisis del proyecto, en la sección 4 se pueden encontrar los requisitos, tanto funcionales como no funcionales, obtenidos durante el desarrollo.

Implementación

Toda la información detallada de la implementación de todas las técnicas mencionadas anteriormente puede ser encontrada en este apartado. Además, acompañando a cada explicación se incluyen partes del código relevantes relativas al método explicado.

En lo relativo al Editor de Terreno se explica en detalle los cálculos realizados para la obtención de la normal en cada pixel en la implementación de la técnica Normal Mapping. Además se incluye una explicación del filtro Sobel utilizado para obtener el mapa de alturas del terreno en un momento determinado. Todo esto esta codificado en el lenguaje HLSL (High Level Shader Language). Éste es un lenguaje de bajo nivel, ejecutado directamente en la GPU. Cuantos más cálculos se hagan en esta parte del código mejor, ya que liberamos al procesador principal, CPU, de ellos. Además, la unidad de procesador gráfico es más rápida, obteniendo así mejores resultados. Por tanto, un buen diseño e implementación de estas técnicas en HLSL es vital para el correcto funcionamiento y alto rendimiento de las herramientas. Todo el código que aquí se incluye ha sido sometido a optimizaciones constantes durante todo el desarrollo del proyecto para conseguir las mejores prestaciones posibles. Dado que una malla de terreno tiene más de 1 millón de vértices, la optimización de todos los cálculos es algo vital. Como prueba de ello se incluyen en la sección Demostración diferentes pruebas que demuestran al lector la importancia de lo aquí explicado.

En la sección de implementación de la Máquina de Estados se hace hincapié en la necesidad de implementar un código lo más sencillo posible. Como prueba de ello se incluye el código relativo a la clase principal de la Máquina de Estados que tan sólo dispone de unas pocas líneas de código. De este modo se consigue uno de los requisitos indispensables mencionados a lo largo de este resumen. Además, se incluyen otras clases más complejas como son las destinadas a la animación propia de cada personaje. Aquí se explica la importancia de la "*Content pipeline*" y el proceso de exportación necesario entre SoftImage ModTool (XSi) y XNA. Éste no es un proceso trivial y ha sido objeto de numerosos problemas durante todo el desarrollo. Las clases propias de XSi para el tratado de los modelos 3D y las animaciones disponen de dos errores que han tenido que ser resueltos en la herramienta para poder conseguir un resultado óptimo. El primer error (*bug*) impedía a un modelo 3D colocarse en una posición del espacio determinada distinta a la inicial. El problema en este caso era que los cálculos de la transformada del modelo realizados en el *shader* que por defecto usa XSi no tenían en cuenta la posición local del modelo, si no la global. Por tanto, el modelo al ser dibujado en pantalla se situaba en la posición en la que fue modelado (normalmente en el origen), ignorando cualquier otra transformada que se produjese durante el código en ejecución. La solución de este error se incluye en el apartado 6.2 junto con todas las demás explicaciones referentes a la implementación de la Máquina de Estados. Por otra parte, la resolución del segundo error también ha sido clave para la correcta consecución del proyecto. En este caso, la funcionalidad que provee por defecto XSi no permite ejecutar diferentes animaciones en instancias del mismo modelo 3D al mismo tiempo. Esto que a primera vista puede resultar no muy relevante, pero para el desarrollo de un videojuego resulta vital. Una manera muy común de "ahorrar" recursos en un videojuego es utilizando el mismo modelo repetidas veces en situaciones en las que la posibilidad de que el usuario se dé cuenta de ello son muy reducidas. Una ampliación de esta técnica es utilizar el mismo modelo pero con diferentes texturas u objetos unidos a él. Este puede ser por ejemplo el caso del público en un estadio deportivo. Pero no sólo debemos pensar en personajes animados. Otros objetos son animados en un videojuego para añadir más realismo a una escena como árboles o nubes. Sin embargo, disponer de un modelo único para cada objeto es un gasto inaceptable en el desarrollo. Con la funcionalidad por defecto que ofrece XSi, todos los personajes de un mismo modelo dispondrán de exactamente la misma animación en un mismo instante. La consecuencia de esto es la perdida completa de realismo en la escena. Este error se debe a que XNA carga una sóla vez cada modelo aunque éste sea después utilizado en diferentes instancias. XSi asocia un tiempo global a cada modelo, otorgando así el mismo tiempo global a cada instancia. Como solución cada instancia o personaje debe guardar su tiempo local. Además el procedimiento *PlayBack()* ha sido sobrescrito por un método de extensión que tiene en cuenta este tiempo local en lugar del global. Así es posible que cada personaje tenga su propia animación en un momento dado.

Para demostrar el buen rendimiento de esta herramienta se ha incluido un test de rendimiento, en el que 32 personajes más el personaje principal son animados al mismo tiempo. En este punto, el lector entenderá más claramente la importancia de solventar los errores mencionados en el párrafo anterior. Con estos errores los 33 modelos aparecerían en el mismo punto, y las instancias de los mismos modelos tendrían las mismas animaciones. El grupo de personajes (*crowd*) es generado de manera totalmente aleatoria, así como su comportamiento cambiando de estado cada cierto tiempo. Los detalles de implementación utilizando clases genéricas están también incluidos en este punto.

Demostración

En este apartado se incluyen fundamentalmente capturas de pantalla de ambas herramientas. Por un lado se muestra la herramienta *Terrain Editor* en sus tres distintas configuraciones. El lector podrá ver a simple vista la mejora que supone cada una de ellas visualmente. Pero además, para demostrar fielmente la mejora que supone una con respecto a la otra, se incluye los *fps (frames per second)* en el momento de cada ejecución. Esta es una medida muy fiable de la cantidad de recursos consumidos por una aplicación gráfica como es nuestro caso.

Para la demostración de la segunda herramienta, *Finite State Machine,* también se incluyen capturas de pantalla. En este caso se ha elegido mostrar cada personaje y las distintas animaciones de uno de ellos como ejemplo. Así mismo, al igual que con la herramienta anterior, se incluye un test de rendimiento para demostrar la implementación óptima de esta. En esta ocasión la prueba consiste en mantener en pantalla un número elevado de personajes animados al mismo tiempo bajo un número de *fps* aceptable.

En este punto hemos demostrado la eficacia individual de cada herramienta. Pero como su nombre indica son herramientas, y por ello no van a ser utilizadas individualmente si no en el conjunto, que será un videojuego completo. Por ello, además de los test propuestos anteriormente se han incluido dos más. El primero es una de las distintas configuraciones posibles en la herramienta *Finite State Machine.* En esta configuración se ha integrado el terreno diseñado con el *Terrain Editor* con los modelos y animaciones. De este modo es posible ver a los personajes en un entorno más real y propio de un videojuego. El personaje principal se moverá por el escenario tal y como lo haría en el juego. Es decir, respeta las diferencias de altura del terreno. Además se ha implementado una cámara de tal modo que siga al personaje principal. Esta cámara dispone de un sistema de “muelle” que posibilita unos cambios de posición mucho más suaves en vista del jugador. Con esto se consigue una mejor inmersión en el juego. La herramienta dispone de un menú de *debug* en el que es posible configurar todas las características mencionadas anteriormente, además de ver datos útiles como el contador de *fps*, las posiciones del personaje actual o la cámara.

El segundo test es la participación de estas dos herramientas en un juego completo e independiente al desarrollo de este proyecto. Este juego llamado *Tremors* (Temblores) utiliza al máximo ambas herramientas, siendo la calidad y deformación en tiempo real del terreno sus mayores atractivos. Se incluyen en este documento capturas que demuestran lo aquí adelantado. La demostración con este juego se limita a las explicaciones y capturas de pantalla incluidas en la memoria. Se ha decidido deliberadamente no incluir el código ya que es una aplicación completa que no forma parte del desarrollo de este proyecto y ha sido utilizada tan sólo para demostrar las posibilidades de las herramientas.

Conclusión: evaluación, conocimiento adquirido y futuras ampliaciones

Este ha sido un proyecto ambicioso, en el que se ha pretendido desarrollar dos herramientas para uso real en un juego profesional basado en *XNA*. Durante todo el desarrollo se ha intentando seguir el *Gannt chart* propuesto al principio del desarrollo. Pero al mismo tiempo, la organización y planificación ha sido flexible adecuando y modificando la tabla con cada iteración. Se ha conseguido que el producto final cumpla con las expectativas de diseño propuestas al principio del desarrollo.

Gracias a este proyecto he podido participar directamente en el desarrollo de un videojuego, involucrándome en las distintas partes de éste. Ahora más que nunca está presente en mí la importancia de la calidad alta que deben tener las animaciones en un videojuego. Pero conseguir esto supone un gran esfuerzo. Distintas partes deben de trabajar juntas: diseñadores y programadores para conseguirlo. Además cada vez los juegos tienden más a un ambiente hiperrealista, siendo la deformación del escenario en el que se desarrolla la acción una parte vital de para conseguirlo. Pero elemento, si no se implementa con cuidado puede suponer el consumo de gran parte de los recursos. Por ello es vital diseñar un terreno con un rendimiento alto.

Estas dos herramientas pueden ser aplicadas de distintas formas. Por una parte, el editor de terreno puede incluir otros elementos como edificios, vegetación, piedras o cualquier otro elemento que se quiera añadir a un posible escenario. De este modo pasaríamos de tener un editor de terreno a tener un completo editor de niveles. Cada elemento introducido en el editor podría ser serializado en un documento XML para que posteriormente el terreno junto con todos sus objetos pudiera ser recuperado o usado en el videojuego. Además de esto existen otras técnicas para conseguir texturas realistas sin necesidad de aumentar la dificultad y el detalle de la superficie. La técnica de mapa de normales (normal mapping) puede ser sustituida por Parallel mapping. Esta técnica se basa en los mismos principios de la anterior pero produce resultados de una más alta calidad, aunque a costa de un mayor consumo de recursos debido al grado de dificultad de los cálculos que son precisos en su implementación. En cuanto a la máquina de estados, es posible implementar más en detalle la técnica de mezcla (blending) entre animaciones. Una mejora importante podría ser la de permitir mezclar animaciones pero sólo en determinadas partes del cuerpo. Así podríamos utilizar para el tronco una animación de apuntar con un arma, mientras que para las piernas podríamos utilizar la de correr. De este modo el catálogo de las animaciones aportadas por los diseñadores se podría hasta triplicar con distintas combinaciones por medio de la programación. Esto supone un gran ahorro en el tiempo de desarrollo de un proyecto. Otra posible ampliación de esta herramienta es la de incluir más contenido dentro de la misma. Debido a que esta aplicación fue ideada para ser usada en cualquier juego, incluir nuevo contenido, ya sean más animaciones, nuevas máquinas de estados, nuevos posibles comportamientos de un personaje existente o incluso nuevos personajes, es muy sencillo.

Una posible mejora muy significante en cuanto a la animación de los personajes se refiere es la de implementar ésta en la GPU. Todo código escrito en HLSL es código que se ejecuta en la GPU y, por tanto, es mucho más rápido. Esta ventaja es posible aprovecharla de muchas maneras. En el caso de la animación, se puede utilizar una textura para codificar y guardar toda la información referente a cada animación. Después, esta información puede ser procesada en el *shader* para producir la animación de una manera más eficiente. Como consecuencia esta técnica permite animar cientos de modelos simultáneamente conservando un número de *frames* por segundo aceptable.

Palabras clave: Editor de terreno, height mapping, normal mapping, Phong model lighting, finite state machine, keyframed animation, XNA Game Studio Framework, máquina de estados.

1. Abstract

My main responsibility on this project is to solve two different common problems when developing a video game. On the one hand, I have researched on the need of games to have realistic scenarios, including the possibility of being modified dynamically during playtime. As a requirement, the tool designed for this purpose should be easy and intuitive to control due to it will be used theoretically by level designers to performance different stages for a game. On the other hand, it is well known that animations are, at the same time, an indispensable feature in a game and a bottleneck during the development. Therefore, the overall aim of this research project is to solve the previously addressed problems in an efficient and satisfactory way. To achieve my goals, two different tools were implemented: *Terrain Editor* tool and *Finite State Machine* tool.

Both tools were designed to be used within the XNA framework, for two different platforms: Windows and Xbox360. In addition to this, either the Terrain Editor or the FSM tools are independent from the game that will be applied on. This means that by just changing the project content (models, animations and textures) the tools can be used in the development of almost any kind of game within the XNA Framework. Nevertheless one specific game was chosen in order of the implemented tools to be evaluated and to demonstrate the capabilities of them. The reader of this paper will find more detailed information about this game in section 3: Background. Furthermore, the tools were used in a second game that takes advantage of the potential of the tools. The main feature of this game, called *Tremors,* is the realism of the deformable dynamic environment. More information about this game can also be found in section 7: Demonstration, including screenshots taken during play time.

Video games development can be considered an art. However, during the creation of a game a great bunch of different things can go wrong, wasting a lot of work, effort, dedication and, why not, money. The tools proposed in this paper were implemented with the idea, not only of minimized risks concerning the terrain creation and animations, but also of using advanced techniques to achieve great performance. The high quality results achieved with these tools can make big difference in the overall game product.

The techniques selected to implement the tools are height mapping, normal mapping, Phong model for lighting and altitude based texture sampling for the Terrain Editor tool and finite state machines with keyframed animations for the second tool. During the reading of this paper, it will be shown to reader the advantages of those techniques and why they were chosen.

2. Introduction

The project was conceived to help developers during the creation of a video game. Bearing this mind, the tools need to satisfy the requirements of the different members involved on this process: programmers, designers, artist, level designers... To do so, some important requirements like efficiency, response time and usability were taken into account during the design process of the tools.

As it was mentioned in the previous section, this project solves two different problems, on the one hand the development of a high quality and dynamically deformable terrain. Advanced techniques were implemented to do so: height mapping and normal mapping, as well with a Phong lighting model and altitude based texture sampling. On the other hand, the development of a bridge tool between the animation software and the game code in order to reduce the bottleneck in this part of the process. This second tool manages animated 3D models created by artist using a finite state machine. Therefore, the content itself is not important for the purpose of the tool. The application is designed to be used independently of the content.

Through this paper, the reader will find a detailed explanation of all the steps involved during the development of both tools and other useful and relevant information. Due to the problem meant to be solved by this project is not unique; each point of this paper will be described twice for both tools. The next section contributes to the understanding of the project's background. It sets the context and prepares the reader for the following points. During these points, the level of detail will be increased and features like UML diagrams and pieces of key code will be added in order to give a clear understanding view of the analysis, design and implementation of the project. The last two sections provide a demonstration of the results achieved with the tools and a main conclusion with reflections of the process and outcomes of the work. The demonstrations provided here are quite important to understand the scope in which both tools are designed to be used.

Other appendices are included for the completeness of this final report. Appendix A includes a list of all references used during the research. While appendix B consists of resources used during the development process of the tools. The third appendix, C, is a brief guide to introduce the reader in the use of the tools, including installation notes and the input list.

3. Background and literature review

3.1 Background

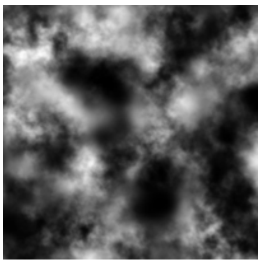
As it was mentioned previously in the Abstract, although the implemented tools are orientated to be used independently from the type of game, the project is based on the development of a real videogame called *The DarkSoul Paradox (DSP)*. It is a 3D futuristic game developed in C# using Microsoft XNA Studio Game. The software will be released, in different chapters, for gamers all around the World using *Xbox Community Games.* Therefore, DSP is a long-term project, which means that any tool developed for its development must be scalable, flexible and non dependent of software versions. In addition to this, the use of free licensed software and resources was a must. Using an existent project has helped me to understand in a more clear way the scope of the problem to be solved, to orientate the functionality in a realistic way and to set all the needs and issues during the development of software. Moreover, the content provided by the group allows me to demonstrate the completeness and correct functionality of the tools.

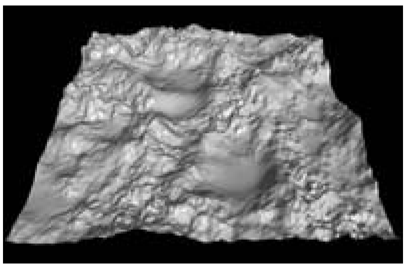
3.2 Literature review

The following point is divided in different sections corresponding with the techniques implemented for each tool: height mapping, normal mapping, Phong lighting and altitude based texture sampling according to the Terrain Editor tool and the finite state machine technique and keyframed animation for the second tool. In each of the following points a description of the technique will be given. If a more exhaustively understanding of them is needed please refer to Section 6: Implementation. All the techniques necessary to resolve the problem addressed previously on this paper will be fully explained in these two sections.

3.2.1 Height mapping

An image used to store values like the surface elevation data is called a **heightmap**. A heightmap contains one channel interpreted as the height of surface in each point. It is visualized as a greyscale image, with black representing minimum height and white representing maximum height. The following images [1] show an example of a





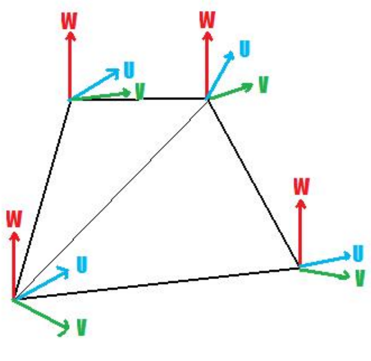
heightmap texture and the same texture converted in 3D mesh. At this point the user should be able to imagine the potential of this technique and why it is commonly used in games. Some industry examples are: Sim City, Battlefield or Far Cry. The initial black and white texture will be modified and saved during execution time in order to allow the terrain to change dynamically.

3.2.2 Normal mapping

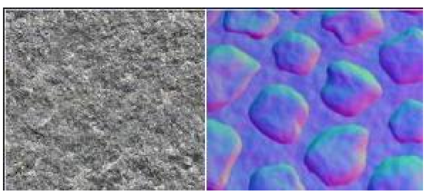
Riemer (2009, p.476) have described **bump or normal mapping** in this way: *"is a technique that gives the viewer the impression of height differences with a large triangle by changing the colour of each pixel in the triangle"*. Therefore, this technique is a way to make a low-poly object look like a high-poly object, without having to add more polygons to the model and making it more expensive to compute. Thus, it will be possible to use this technique to make the terrain or any other surface, look more realistic without consuming too much processing power. The *normal mapping* technique has multiple benefits. Supporting evidence appears in a study by Ernst (cited in Jun et al, 2005, p.166):

> *"1) A high level of visual complexity in a scene, without adding more geometry. Thus, this technique allows the game to have high quality graphic performance without overloading the CPU and GPU capacity.*
>
> *2) Simplified content authoring, because surface detail can be encoded in textures as opposed to requiring designing highly detail 3-dimensional models.*
>
> *3) The ability to apply different normal maps to different instances of the same model to give each instance a distinct surface appearance. Therefore, this technique will be very useful for rendering multi-textured models like a terrain."*

The purpose of using this technique is to achieve good rendering quality without incrementing the number of polygons. The normal of each texel is represented by a three dimensional vector that stores the RGB values. These normals will be used afterwards to compute the lighting calculations in the shader. These normals are stored in a texture space or tangent space coordinate system (see left image [2]). The shader will create the vector W for the texture space coordinate system by using the normal. Then the desired vector V will be calculated by taking the cross-product of W and U.



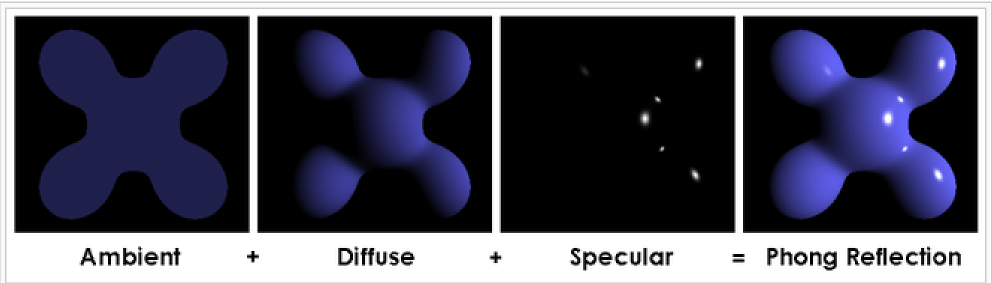
To do this, two textures will be needed: one for the desired texture itself (rocks, soil, grass...) and another one for the normal map of that texture.



An example of these textures can be seen in the right image [2]. The left one corresponds to a stone texture, the right one is the normal map used with it. The normal map texture can be created using a graphics editing program like Adobe Photoshop.

3.2.3 Lighting: Phong model

Phong shading is a model of local illumination. Glassner, A. (1997, p.83) accurately defined this shader as: “Phong shading means interpolating surface normals to find intermediate normals that we then evaluate with respect to the light source to find a colour for that point.” Therefore, this technique describes the way a surface reflects light as a combination of ambient, diffuse and specular colours. It is commonly used among developers due to it is a cheap way to calculate lighting with high quality results. The following image [3] demonstrates the combination and result of the different types of lighting.7



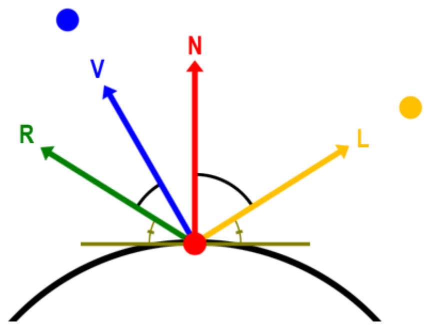
The Phong lighting model provides an equation for computing the shading value of each surface point (I_p):

$$I_p = k_a i_a + \sum_{m \in \text{lights}} (k_d (L_m \cdot N) i_d + k_s (R_m \cdot V)^{\alpha} i_s).$$

i_a : ambient light intensity
i_d : diffuse light intensity
i_s : specular light intensity

K_a: ambient reflection constant
K_d: diffuse reflection constant
K_s: specular reflection constant
α : shininess constant of the material

The following diagram[4] represents the previous formula:

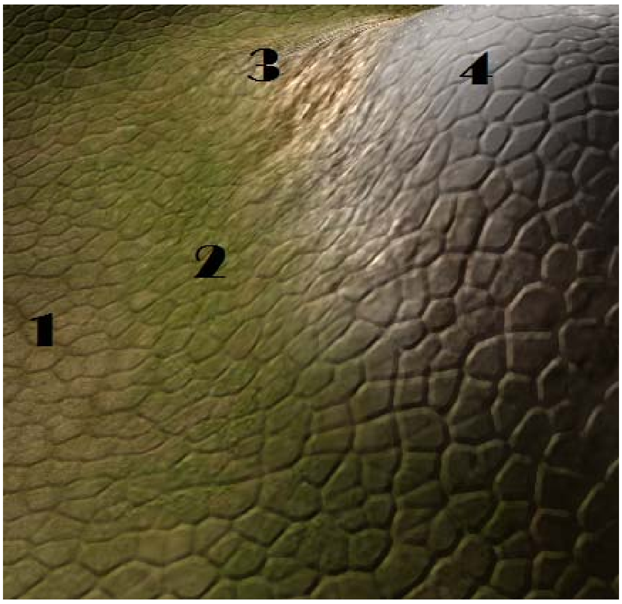


L (**vector to light**): direction from the point on the surface toward each light source.
N (***surface normal***)
V (***view vector***)*:* direction pointing towards the viewer.
R (**reflection vector**): direction that a perfectly reflected ray of light would take from this point on the surface.

The specular part of the equation is large only when the view vector (V) is aligned with the reflection vector (R). The angle between V and R is ß. The more V is aligned with R, the brighter the specular light should be. Therefore, the value of cos(ß) can be used to describe the specular reflection. Additionally to characterize shiny properties a 'α' power is used. When α is large, in the case of a mirror reflection, the specular component will be small, because any viewpoint not aligned with the reflection will have a cosine less than one which rapidly approaches zero when raised to a high power.

3.2.4 Altitude based texture sampling

This technique allows the Terrain Editor to place automatically the textures depending on the height of the terrain at each point. To do this, the textures are map to certain range of heights, from the lower to the higher one. In addition to this, the textures next to the border values are blended in order to achieve a gradual effect from one texture to another. This increases the realism in the scenario by avoiding sudden changes. It would be possible to specify other parameters to determine which texture to use. For example, not only the height (y) by the x and z position to create a specific element of the terrain like a lake or sea. This technique offers a great amount of different possibilities. The screenshot at the left shows four different textures applied depending on a height range as example. It is possible to distinguish



perfectly the blending between textures in the limit of each range.

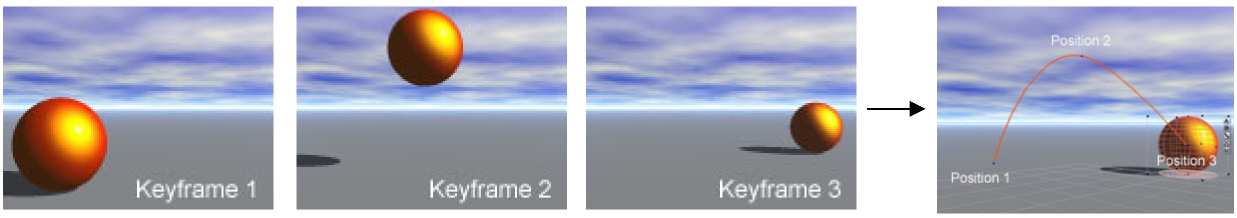
3.2.5 Finite State Machine and keyframed animation

A **finite state machine** is a model of behaviour composed of states, actions and transitions between states. It could be defined also using mathematical terms: *"FSM is a proper function that maps input sequences into the set of output sequences"* (Zharikova, 2002). In the specific context of this project the input could be directly the keyboard or the gamepad. Moreover, in other words, the finite state machine interprets this input as actions. Depending on the current state this action will cause a change to another state.

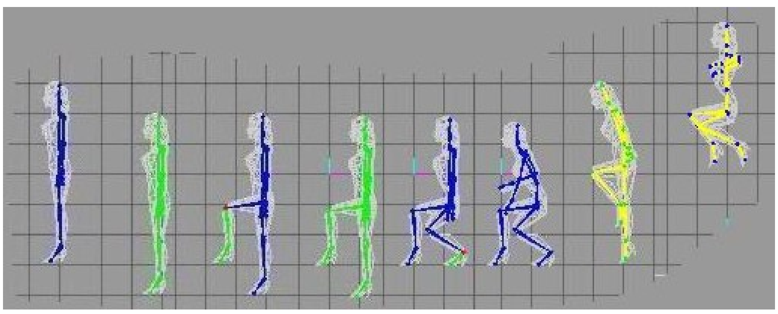
Finite state machines are widely used for animations and artificial intelligence because of their advantages: simplicity, flexibility and easy maintenance. It allows the implementation of complex behaviours by encapsulating simple FSMs as single states of other more complex machine. To develop a finite state machine the first step is to design a state diagram including states, actions and transitions. Once the diagram is reviewed, and it is sure that meets all the requirements, it can be coded. All the diagrams designed for this project can be found later on this paper.

There are two main animation techniques used during a videogame development: keyframed and skeletal animation.

The first technique, **keyframed animation**, allows creating an animation based on snapshots of a sequence. As it is shown on the image [5] below, a ball can be basically animated by using just three keyframes:

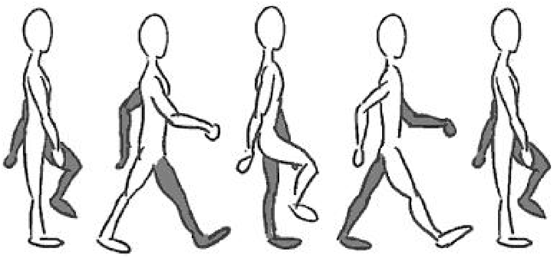


With **skeletal animation** each vertebrate character is a hierarchy of bones or rigging. Each of the bones has a transformation matrix: position, scale and orientation. It also has a parent bone which manages the rest of the hierarchy. The full transform of a child node is the product of its parent transform and its own transform. As the character is animated, the bones change their transformation over time; therefore they control the



deformation of the character mesh. The following image (Lobão, 2008) shows an example of this type of animation:

The **keyframed animation** is the technique chosen to implement the locomotion on this project. This technique used for character animation will consist on exporting, from the modelling software to the XNA project, a static mesh per each keyframe of the animation wanted (See an example of 4 different keyframes on the left image (Riemers, 2008)). Once all the static meshes are stored, the complete animation will be obtained by interpolating each pair of keyframes. Interpolation is a mathematic method of constructing new data points (intermediate animations) within the range of a discrete set of known data points (keyframe animations). The result is a smooth model animation. All models used in the development process of this project are implemented using *Autodesk Softimage ModTool* software. This 3D computer graphics application for producing 3D computer graphics, 3D modelling, and computer animation has been used in important industry titles like Metal Gear Solid, Half Life 2 or Resident Evil.



On the one hand, the reason of this choice is because this technique is faster, due to all the interpolation calculates are done automatically by XNA. On the other hand, a drawback of this method is that consumes lots of storage resources in order to store in memory a mesh per each frame. Nevertheless, this will not be a problem considering the final amount of data in this specific project.

Furthermore, a content pipeline will help to integrate the exported models into the Finite State Machine tool. The reader would fine more details about the Softimage ModTool (XSi) specific classes and its XSIXNARuntime pipeline in the implementation section.

4. Requirements specification

In the next lines of this paper the reader can find a list of requirements: external, functional and non functional.

4.1 External Interface Requirements

Tool installer package, one per each tool
PC + Keyboard. Mouse not used.
Xbox gamepad controller [optional, although highly recommended]

4.2 Functional requirements

Terrain Editor tool	
Tool initialization	The user should be able to access to the main menu when starting the tool.
Menu navigation	When a menu is displayed, the user should be able to navigate through the different options and enter in each one. Also, going back to the previous screen must be necessary in all cases. The input used is either the keyboard or an Xbox Gamepad usb controller connected to the PC.
Camera movement	Using the appropriate input, the user should be able to rotate, move and change the altitude of the camera within specific values.
Switch between shaders	While editing the terrain, the user should be able to switch between the different light modes (i.e. shaders) by pressing the specific key or button.
Terrain modification	While editing the terrain, the user should be able to modify the orography of the surface by lowering or raising it. To do this the specific key or button must be press.
Changing brush type	In order to modify the terrain, a bunch of different brushes are provided. The user should be able to toggle between them. The result on the terrain orography will vary depending on the brush shape.
Changing brush properties	While editing the terrain, the user should be able to change the brush properties (size and strength). These changes should be reflected when modifying the terrain using the selected brush.
Saving the terrain	The user should be able to save the modifications done in the terrain, either by pressing the specific key or button, or through the pause menu.
Loading the terrain	While being on the main, the user should be able to switch between the different light modes (i.e. shaders) by pressing the specific key or button.

Finite State Machine tool	
Tool initialization	The user should be able to see displayed the first default character and animation when starting the tool.
Navigate through the different states	By using the specific input, the user should be able to navigate through the different states of the machine according to the behave diagrams on this paper.
Change to a different character	It should be possible for the user to cycle between the different characters implemented in the finite state machine.
Camera movement	With the appropriate input the user should be able to rotate the camera and change the altitude between specific values.

4.3 Non functional requirements

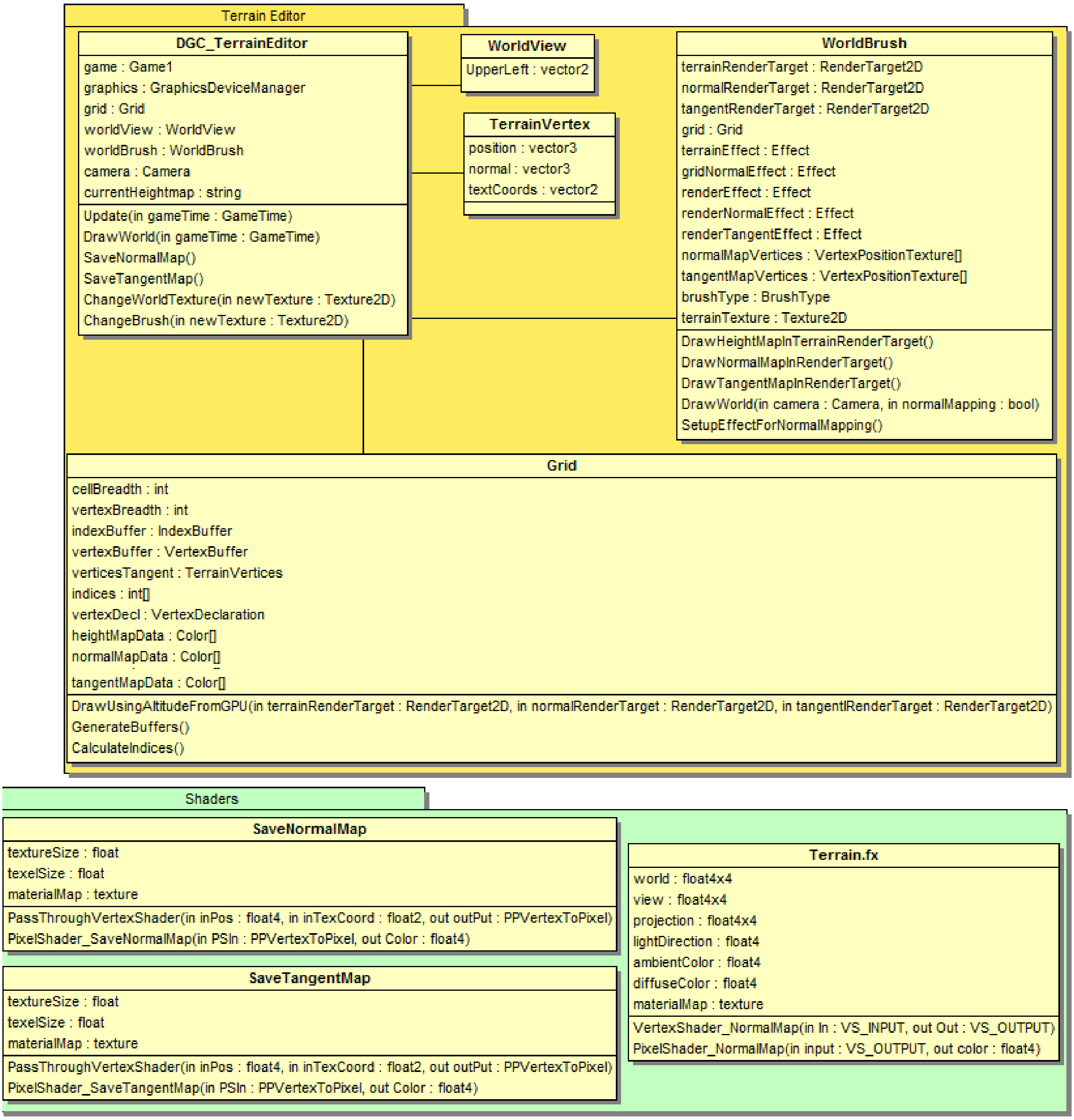
Performance: both tools run at 60 fps.
Response time: high response time (low latency)
Documentation: each tool has the appropriate installing and input instructions.
Constraints: develop the tools using free software.
Upgradability: both tools must be easy to upgrade with new illuminating or terrain techniques in one case and with more animations and characters in the other.
Dependability: the installing package can be executed in any PC with Windows O.S, even when the XNA Framework is not installed on it.

5. Analysis and design

In the following section the reader will find UML and finite state machine diagrams to help the understanding of the design process. At this point it is worth to remember that both tools were designed to be used in different projects. Thus, the design has to be as simple and as free of dependencies as possible. This is one of the reasons why the main classes of each tool are encapsulated in Game Components.

5.1 Terrain editor

The terrain editor tools can be divided in three blocks: Engine, which contains the classes relative to the control of the input and the camera. The second block is the component that manages the menu, and the last block corresponds to the terrain itself. This third block is the one that will be analysed in following lines.

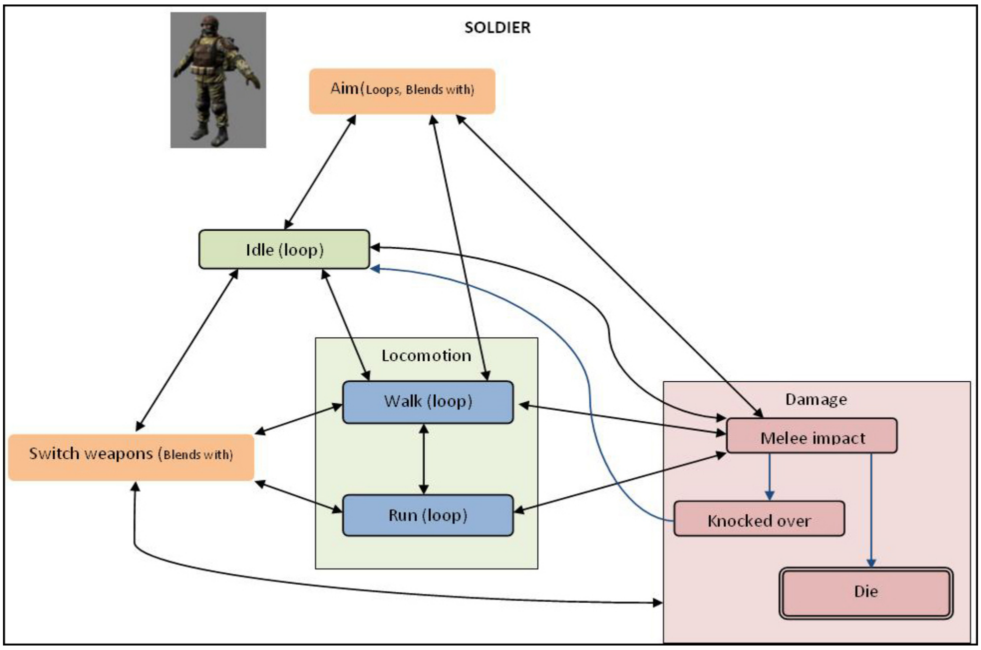


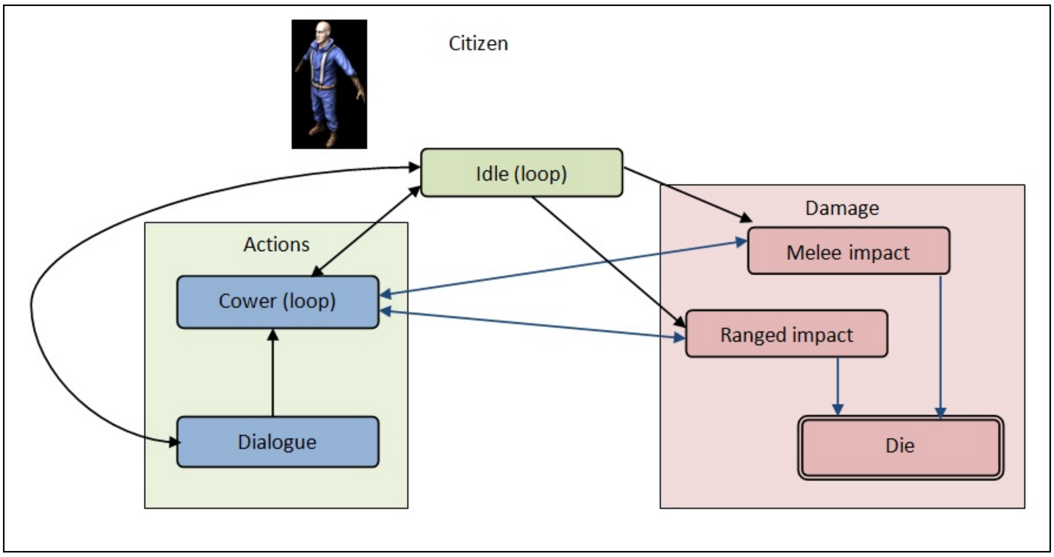
UML diagrams:

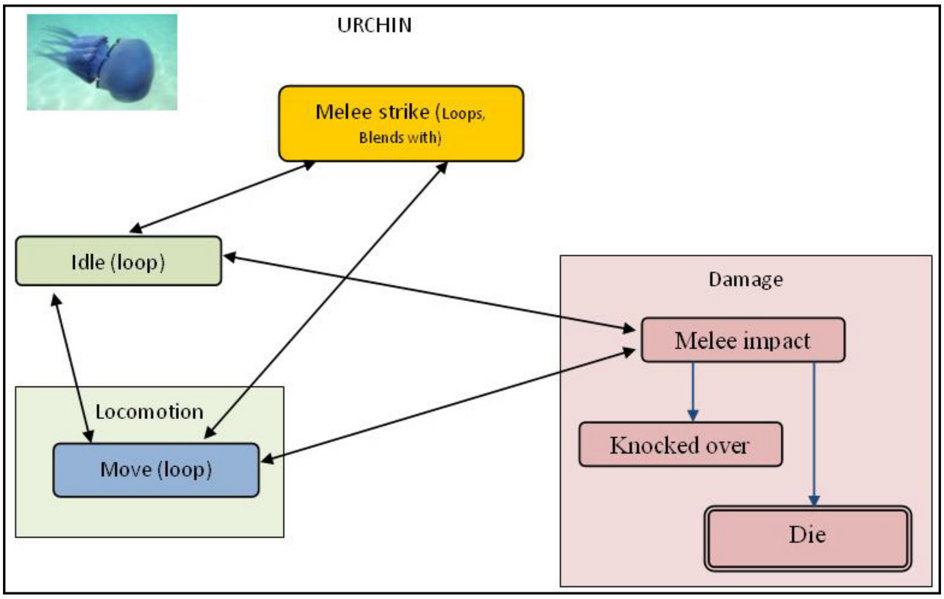
The Terrain tool is designed as a Drawable Game Component, in order to be used easily in other projects. This tool also uses other Components: GC_InputManager, Camera and DGC_MenuManager. Each of these components can be substituted by any other which implements the same functionality.

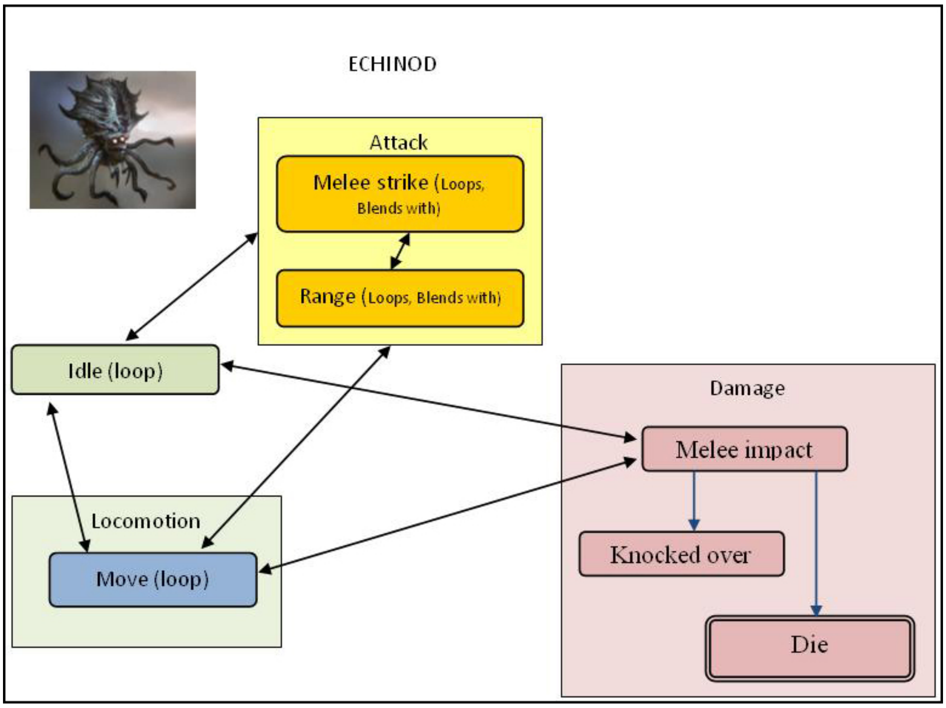
5.2 Finite State Machine

Next, the reader will find the finite state machine diagrams for four different characters: Soldier, Citizen, Echinod and Urchin. The tool implements these diagrams as finite state machines and character classes. However, this is done just as example in order to demonstrate the functionality of it. It is possible to change the content of the tool, i.e. the finite state machines, diagrams, characters, models, depending on the project in which this tool is going to used in.



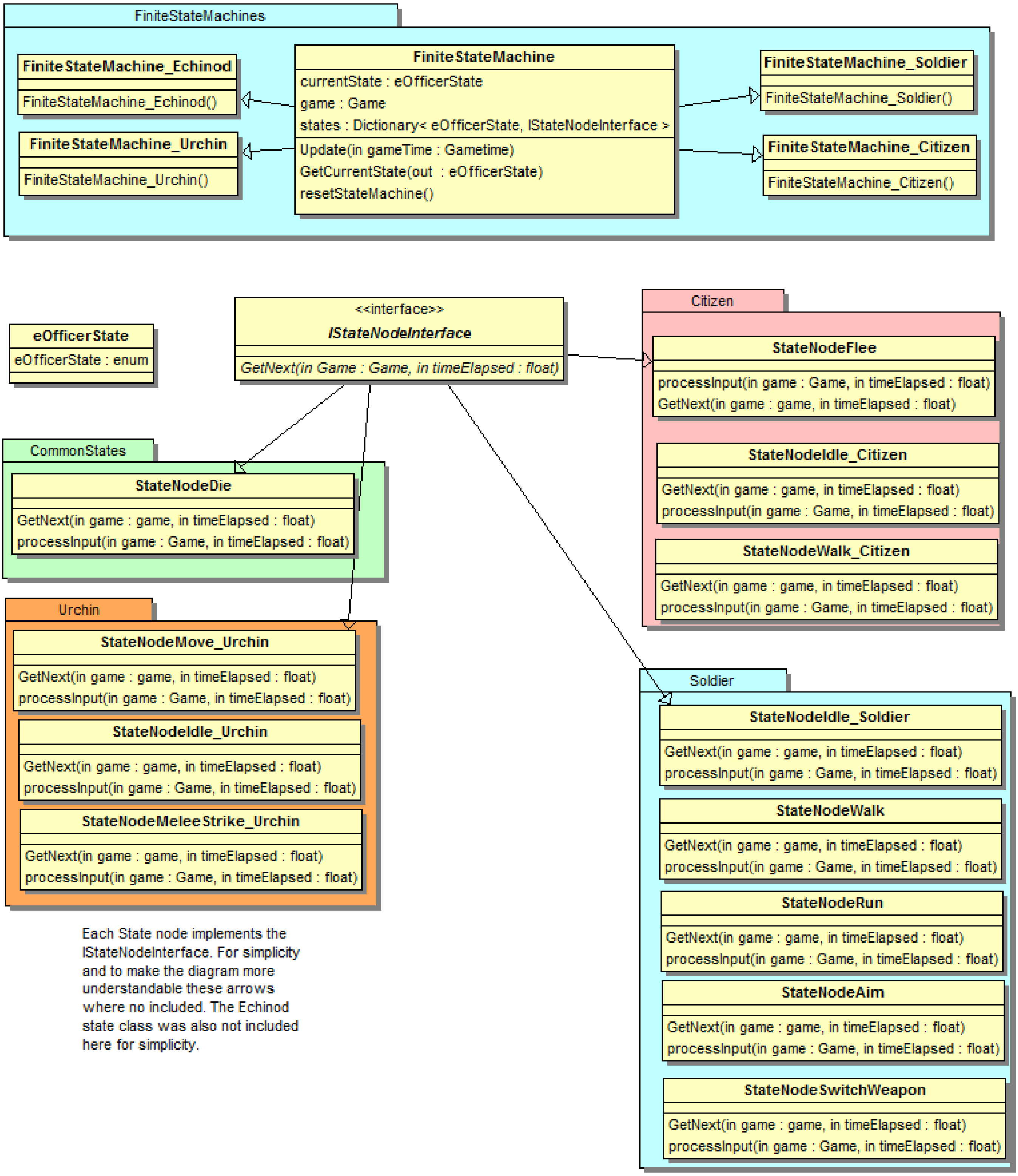


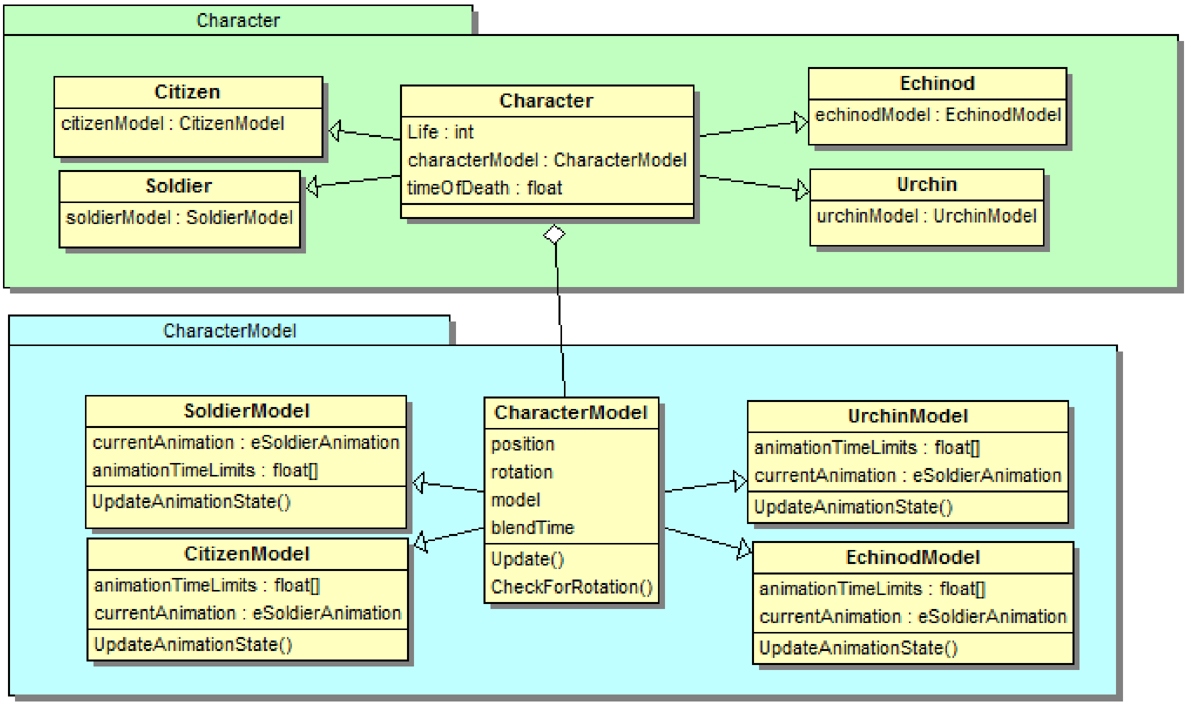




These diagrams show the finite state machine implemented for the project, thus all the possible locomotion animations within the particular game. The diagrams also define which animation (state) is accessible from other states. Each state represents a high-level behaviour or movement. Using the keyboard/gamepad input it is possible to change the state by appropriate actions. The transitions are defined in the FSM algorithm. The only final state possible to access is *Die*; once this state is reached the player will not possibly make any other transition until the finite state machine is reset. The finite state machine of the main character could be check at any moment during runtime. Through the appropriate menu, the list of the states will be displayed. The accessible states at that certain moment will be highlighted.

UML diagrams:





6. Implementation and testing

6.1 Terrain Editor

The Terrain Editor tool uses a bunch of advanced techniques together to achieve both realistic and efficient results. The more important techniques and the ones explained here are: normal mapping, height mapping, Phong lighting model, altitude based texture sampling and heightmap collision detection.

As it was explained in previous points all these techniques are used nowadays in the games industry. The implementation of them uses both C# and HLSL (High Level Shader Language). Due to the great amount of data involved and the complexity of the maths, in the shader is done the biggest part of the calculations. The reader will be able to see the huge difference that represents using the shader (GPU) or the CPU to make the maths. This is the reason why the user can choose three different modes in the Terrain Editor.

6.1.1 Normal mapping

The more important file for the implementation of this technique is the vertex shader called *Terrain.fx*. It is in this file where the normal, tangent and bi-tangent for each pixel are calculated. The detailed process is explained in the following lines. In this shader, three different techniques are implemented:

1. The *Unlit* method: no light calculations are made here, only the different texture placement depending on the height.
2. *Normal mapping* method: the normal map technique and the Phong lighting model are implemented here,
3. *Normal mapping vertex texture* method: the difference with the previous technique is that the altitude calculations are done in the shader, i.e. the GPU, instead of the CPU. Thus, this is the more effective method and the one which has the best frame rate.

The three methods are available in the tool and accessible through the specific input button/key. The user will be able to switch between techniques in order to see the different performances achieved. A demonstration of the differences between can be found in the next section.

These methods are based on the normals of each pixel. There are two sets of normals used with normal mapping, the **model normals** and the **image normals**. The image normals live in 'tangent space' and are generally blue tinted, with ‘X’ being stored in the red channel, ‘Y’ being stored in the green channel, and ‘Z’ in blue channel. The ‘Z’ axis points towards the viewer causing the image to be generally blue as shown in the previous diagram and picture.

The model normals are used in conjunction with the **tangent** and **bi-tangent** to transform the light from 'world space' into 'tangent space'. This is necessary because the normal map will be drawn on polygons with different orientations, but the normal map itself will use the same (viewer facing) data, so the lighting calculation for each polygon needs to take into account the orientation of the polygon as well as just the normal map on it. We can calculate the Tangent Space to Object Space matrix based only on the normal, by assuming that all vertices are arranged on an XY (in tangent space, not world space) aligned grid. This means that the bi-tangent will always be aligned with the X axis. We can therefore create an X axis aligned 'dummy bi-tangent'. It will not be at 90 degrees to the normal unless the normal is pointing straight up, but it will be on the same XZ (tangent space) plane as the real bi-tangent. This means that the cross product of the 'dummy bi-tangent' and the normal will be equivalent to the tangent.

Why do we transform the light from 'world space' into 'tangent space', instead of the reverse (transforming the normals from their 'blue tangent space' into proper 'world space')? Due to this way is more efficient. The shader will only handle a few lights, typically, whereas a 1024*1024 normal map would have more than a million normals in it. So we invert the problem and put the lights into 'world space', in the vertex shader, using the model normal, tangent, and bi-tangent to form a 3x3 rotational matrix, which we apply to the light.

Terrain.fx

```
** Vertex Shader **
// Tangent Space to Object space matrix
    In.normal                 = normalize( In.normal );
    float3 dummyBiTangent     = float3( 1.0f, 0.0f, 0.0f );
    float3 tangent            = cross( In.normal, dummyBiTangent );

// Now that we have the tangent, we can determine the real bi-tangent
    float3 biTangent          = cross( In.normal, tangent );

// Now we are all set to create and apply our 'tangent to object'
rotation matrix
    float3x3 tangentToObject;
    tangentToObject[0]        = biTangent;
    tangentToObject[1]        = tangent;
    tangentToObject[2]        = In.normal;

    Out.lightDirection        = mul( tangentToObject, lightDirection );

** Pixel Shader **

// The color describes the direction of the normal vector. Vector T and
Vector W
    float3 normal             = (normalColor - 0.5f) * 2.0f;
    normal                    = normalize( normal );
```

The calculation of the normal is made in the *SaveNormalMap.fx* shader. In the pixel shader of this class, a **Sobel filter** for edge detection is applied to the height map texture. The technique, widely used in image processing, is based on the use of specific value filters in different directions, along x-direction and y-direction concretely. At each pixel, the vector is oriented to the biggest intensity difference and its magnitude is the grade of this difference. Therefore, the result of this process is the normal vector in each pixel. This normal will be used by the light calculations afterwards in the *terrain.fx* shader.

SobelX (edge-detection along x-direction):

```
  1  2  1
  0  0  0
 -1 -2 -1
```

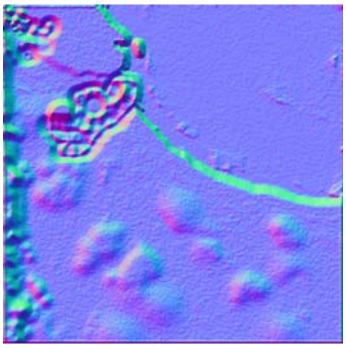
SobelY (edge-detection along y-direction):

```
-1  0  1
-2  0  2
-1  0  1
```

SaveNormalMap.fx

```
float deltaX = topRight + (2 * right) + bottomRight - topLeft –
                     (2 * left) - bottomLeft;

float deltaY = bottomLeft + (2 * bottom) + bottomRight - topLeft -
                     (2 * top) - topRight;

// Build the normalized normal
float reciprocal = 1.0f / normalStrength;
float3 normal    = normalize( float3( deltaX, deltaY, reciprocal ) );
```

An example of the result of the filter can be seen on the left image. This texture is calculated in real time with each modification and it is stored in a render target. The render target is read by the terrain shader afterwards to calculate the light direction depending on the normal. To increase the efficiency of this method, the normal texture is only saved in the render target, thus no physical file is needed. In order to implement the deformable capability of the terrain, every time the surface is changed, the texture is changed appropriately and restored in the render target.



The following chunk of code corresponds with the functionality previously explained. The code belongs to the UpdateTerrain method in `DGC_TerrainEditor.cs`.

```
// End of modifications -> Save world and load it
if( this.changingTerrain )
{
  // Draw the new normalmap in the render target
  this.worldBrush.DrawNormalMapInRenderTarget();

  // Replace the old texture by the new one(resolvedTexture)
  IGraphicsDeviceService graphics = this.game.Services.GetService();
  GraphicsDevice         device   = graphics.GraphicsDevice;

  device.SetRenderTarget( 0, null );
  Texture2D resolvedTexture   = worldBrush.terrainRenderTarget.GetTexture();
  this.worldBrush.ReplaceTerrainTexture(resolvedTexture);
  this.changingTerrain        = false;
}
```

6.1.2 Height mapping

The terrain itself is stored in **vertices** in order to display it. The 1024*1024 texture spreads over 512 square meters, with a vertex placed every 50cm.

```
this.vertices = new TerrainVertex[(this.vertexBreadth) *
(this.vertexBreadth)];
this.indices  = new int[this.cellBreadth * this.cellBreadth * 6];

// Create each vertex and calculate its position
float halfBreadthOffset = this.cellBreadth / 2 * CELL_SIZE;
for (int z = 0; z<this.vertexBreadth; z++)
{
    for( int x = 0; x<this.vertexBreadth; x++ )
    {
       int index  = (z * this.vertexBreadth) + x;

       // Position
       this.vertices[index].position    = new Vector3( x * CELL_SIZE,
0.0f, z * CELL_SIZE );
       this.vertices[index].position.X  -= halfBreadthOffset;
       this.vertices[index].position.Z  -= halfBreadthOffset;

       // Vertices are arranged in rows of 'x'
       this.vertices[index].normal   = new Vector3( 0.0f, 0.0f, 1.0f );

      // UV coordinates
      this.vertices[index].texCoords = new Vector2( x, z ) / cellBreadth;
   }
}
```

Then, the heightmap texture is read and the altitude of it vertex modified according to the colour.

```
Texture2D colorAndHeightMap         = terrainRenderTarget.GetTexture();
colorAndHeightMap.GetData< Color >( this.heightMapData );

// Position
int vertexIndex   = zVertextOffset + x;
int textureIndex  = zTextureOffset + startX + x;
this.vertices[vertexIndex].position.Y     =
(float)(this.heightMapData[textureIndex].A) * HEIGHT_MAP_SCALE;

// Texture coordinates
this.vertices[vertexIndex].texCoords.X    = (x * VERTEX_UV_INTERVAL) +
(startX * VERTEX_UV_INTERVAL);
this.vertices[vertexIndex].texCoords.Y    = (z * VERTEX_UV_INTERVAL) +
(startZ * VERTEX_UV_INTERVAL);
```

This is only the case where the altitude is computed using the CPU. In a more effective way, the height can be calculated in the Vertex Shader by the GPU in the following way:

```
//-------------------------------------------------------------------------
VS_OUTPUT VertexShaderWithVertexTexture( VS_INPUT In )
{
   float2 heightMapCoordinates = (In.position.xz + heightMapOffset) /
(HEIGHTMAP_SIZE * 0.5f);

   float4 heightColor   = tex2Dlod ( heightMapSampler, float4(
heightMapCoordinates, 0, 0 ) );
   float height         = heightColor.a;
   In.position.y        = height * maxHeight;
   […]
}
```

The process has been optimized to only draw 128*128 = 16,384 vertices at a time out of the 1024*1024=1,048,576 vertices available from the heightmap.

In order to **modify the terrain**, the selected brush texture is added or subtracted to the heightmap texture. The brush object has two parameters: size and strength. Depending on this, the amount of modification will vary. The user can set these parameters and select a brush type dynamically during the terrain editing. The reader of this paper can find a sample of all the possible kind of brushes in Appendix C: Tools Guide. This tool was implemented to create a terrain by editing a heightmap. Thus the modification of the terrain is made by the user. However this is a powerful technique that can be automated. Moreover, this method can be used to simulate a crater after an explosion in the ground, steps in the snow or any kind of terrain modification that the reader can think about. The code is shown in the following lines:

```
public void DrawSprite( bool up )
{
   Vector2 viewPortSize = new Vector2( this.terrainTexture.Width,
this.terrainTexture.Height);
   Vector2 brushSize       = new Vector2(
this.brushType.GetCurrentTexture().Width,
this.brushType.GetCurrentTexture().Height);

 this.terrainEffect.Parameters[ "ViewportSize" ].SetValue( viewPortSize );
 this.terrainEffect.Parameters[ "BrushSize" ].SetValue( brushSize );
 this.terrainEffect.Parameters["BrushStrength"].SetValue( this.strength );
 this.terrainEffect.Parameters["MatrixTrans"].SetValue( Matrix.Identity);
 this.terrainEffect.Parameters["terrainTex"].SetValue( this.terrainTex );
 if( up ) // Add the spritebatch = raise terrain
 {
     this.terrainEffect.CurrentTechnique=
this.terrainEffect.Techniques["SpriteBatch"];
 }
 else
 {
```

```
     this.terrainEffect.CurrentTechnique     =
this.terrainEffect.Techniques[ "InvertSpriteBatch" ];
 }

 this.terrainSpriteBatch.Begin( SpriteBlendMode.Additive,
SpriteSortMode.Immediate, SaveStateMode.SaveState );
 {
    this.terrainEffect.Begin();
    {
       this.terrainEffect.CurrentTechnique.Passes[ 0 ].Begin();
       {
            CalculatePosition();
            this.terrainSpriteBatch.Draw(
                  this.brushType.GetCurrentTexture(),
                  new Rectangle( (int)(position.X), (int)(position.Y),
(int)this.size, (int)this.size ),
                  new Color( 1.0f, 1.0f, 1.0f, this.strength ) );
       this.terrainEffect.CurrentTechnique.Passes[ 0 ].End();
       }
    this.terrainEffect.End();
    }
 }
 this.terrainSpriteBatch.End();
}
```

After each modification the light is recalculated (in case that the normal mapping technique is active). The user will notice how the incidence of the light changes in the surroundings of a zone where the terrain has been changed.

```
DGC_TerrainEditor

//--------------------------------------------------------------------------
private void UpdateTerrain( GameTime gameTime, IInputManagerInterface
inputManager )
{
  […]
  // End of modifications -> Save world and load it
  bool enoughTime = ( timeElapsed - this.timeLastChange ) > 0.5;
  if( ( this.changingTerrain ) && ( enoughTime ) )
  {
      // Draw the new normalmap in the render target
      this.worldBrush.DrawNormalMapInRenderTarget();
      // Replace the old texture by the new one(resolvedTexture)
      device.SetRenderTarget( 0, null );
      Texture2D resolvedTexture = terrainRenderTarget.GetTexture();
      this.worldBrush.ReplaceTerrainTexture(resolvedTexture);
  }
}
```

6.1.3 Phong lighting

The formula that describes the Phong equation is implemented in the normal mapping shader. Just as a reminder this formula is:

$$I_p = k_a i_a + \sum_{\mathrm{m} \in \mathrm{lights}} (k_d (L_m \cdot N) i_d + k_s (R_m \cdot V)^{\alpha} i_s).$$

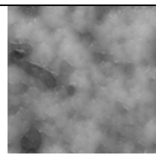
The piece of code which implements the Phong model is:

```
float       specularPower     = 32;
float       shininess          = 0.3f;
float4      ambientColor      = float4( 0.20f, 0.17f, 0.15f, 0.0f );
float4      diffuseColor      = float4( 1.0f,  1.0f,  1.0f,  1.0f );

** Vertex Shader **
 Out.lightDirection            = mul( tangentToObject, lightDirection );

 // Calculate the view direction, from the eye to the surface.
 float3 eyePosition            = mul(-view._m30_m31_m32, transpose(view));
 Out.viewDirection             = Out.worldPos - eyePosition;

** Pixel Shader **
// Phong lighting equation ----------------------------------------------

 // diffuse component: (N.L)
 float diffuse      = saturate( dot( normal, lightDirection ) );

 // R = 2 * (N.L) * N - L
 float3 reflect      = normalize(2 * diffuse * normal - lightDirerction);

 // R.V^n
 float specular      = pow( saturate( dot( reflect, viewDirection ) ),
specularPower ) * shininess;

 // I = A + Dcolor * Dintensity * N.L + Scolor * Sintensity * (R.V)^n
 float4 lightFactor = ambientColor + (diffuse * diffuseColor) + specular;

 return float4( lightFactor.rgb * texColor.rgb, 1.0f );
```

The “normal” and “lightDirection” variables were the result of the normal mapping technique. The result of this function is the specific colour of the processed pixel once the normal mapping technique and the Phong model lighting was applied.

6.1.4 Altitude based texture sampling

Depending on the current altitude of each pixel one texture or another is used. To achieve a more realistic looking the different textures are blended in each "height border" where the textures intersect. To do so, a weight system is used in the *terrain.fx* shader. A weight is calculated in each pixel of the terrain for each single texture. In this case a vector 4 was used due to the terrain uses for different textures (sand, grass, rock and snow). The weight of each texture is stored in the four coordinates. Then every weight coordinate is multiple by its texture while calculating the final colour of the pixel. The range of heights and textures used in the terrain are:

Height (%)	Texture	Sample
[0 - 0.3]	Sand	
[0.3 - 0.6]	Grass	
[0.6 - 0.9]	rock	
[0.9 - 1]	snow	

These textures and their correspondent normal textures can be found in:

...\Terrain\Content\Textures\TextureNormals

The key code of this technique can be seen below:

```
//-----------------------------------------------------------------------------
VS_OUTPUT VertexShader( VS_INPUT In )
{
  float4 TexWeights = 0;

  TexWeights.x      = saturate( 1.0f - abs( height - 0 ) / 0.2f );
  TexWeights.y      = saturate( 1.0f - abs( height - 0.3 ) / 0.25f );
  TexWeights.z      = saturate( 1.0f - abs( height - 0.6 ) / 0.25f );
  TexWeights.w      = saturate( 1.0f - abs( height - 0.9 ) / 0.25f );
  float totalWeight = TexWeights.x+TexWeights.y+TexWeights.z+TexWeights.w;

  TexWeights         /=totalWeight;
  Out.textureWeights = TexWeights;

  return Out;
}
```

```
//---------------------------------------------------------------------------
float4 PixelShader( in float4 uv : TEXCOORD0, in float4 weights :
TEXCOORD2 ) : COLOR
{
  float4 sand          = tex2D( sandSampler,   uv );
  float4 grass         = tex2D( grassSampler,  uv );
  float4 rock          = tex2D( rockSampler,   uv );
  float4 snow          = tex2D( snowSampler,   uv );
  float4 finalColor    = (sand * weights.x) +
                              (grass * weights.y) +
                              (rock * weights.z) +
                              (snow * weights.w);

  […]

   return finalColor * ( lightcalculation );
}
```

All non-related parts to texture sampling have been removed from the previous code to make it easier for the reader.

6.1.5 Heightmap collision detection

Although the explanation of this technique is inside the Terrain section, the current implementation of it belongs to the Finite State Machine tool. There is a good reason for this; one of the demonstration tests implemented for the Finite State Machine is the integration of a terrain on it. The Terrain Editor is designed as a Drawable Game Component, as it was mentioned previously on this paper. This makes it very easy to use the tool in other projects, as it is demonstrated in the integration of it in the Finite State Machine tool. As a Component, it has a Service that allows other Components and classes interact with it. Some basic communication, between the Terrain Editor and the game where it is used, has to be created through the Service. For example, the game should provide the Terrain Editor with the position where the terrain needs to be deformed, plus other parameters like the strength and width of the modification. In addition to this, the Terrain Editor needs to provide the game with the height of a certain position. This position could be where a character or an object is standing, for example. The game will use this information to detect the collision with the heightmap and act consequently, place the character in the correct position in the space, or request a deformation in certain point due to a collision of a projectile with the ground.

The `CheckForLocomotion` function in `CharacterModel` is responsible for the update of the current position depending on the input and the height of the terrain on that point. A critical damper (`FloatCriticalDamper damperAltitude`) is used to avoid sudden changes in the altitude, making the locomotion smoother to the user eyes. The critical damper class is a spring that calculates velocity and delta values to compute the changes of a certain value, in this case the altitude of the main character. This type of objects is widely used in the `Camera` class, again to achieve smoother movements. The use of it and the calculation of the locomotion can be found in the following lines:

```
//--------------------------------------------------------------------------
public void CheckForLocomotion( Camera camera, float speed )
{
  Vector3 forward = Vector3.Normalize(new Vector3((Math.Sin(this.rotation),
                      0.0f, (float)Math.Cos(-this.rotation)));
  if( inputManager.GetPushedState( eInput.Rotate ) )
  {
      CheckForRotation( inputManager, gameTime, camera );
      this.position += forward * speed * this.thumbstick  * timeElapsed;
      this.damperAltitude.targetValue = terrain.CalculateHeightInPoint(
this.position ) + ALTITUDE_OFFSET;
      this.damperAltitude.Update( timeElapsed );
      this.position.Y = this.damperAltitude.CurrentValue;
  }
}
```

The height of the desired point in space is calculated in the terrain:

```
DGC_Terrain

//--------------------------------------------------------------------------
public float CalculateHeightInPoint( Vector3 position )
{
  return this.terrain.CalculateHeightInPoint( position );
}


Terrain

//--------------------------------------------------------------------------
public float CalculateHeightInPoint( Vector3 newPosition )
{
  Float   centre        = DGC_Terrain.GRID_CELL_BREADTH / 2;
  Vector2 texPosition   = new Vector2();

  texPosition.X = centre + (WORLD_TO_TEXTURE_SCALE * newPosition.X);
  texPosition.X = MathHelper.Clamp( texPosition.X, 0.0f,
WorldView.WORLD_SIZE );

  texPosition.Y = centre + (WORLD_TO_TEXTURE_SCALE * newPosition.Z);
  texPosition.Y = MathHelper.Clamp( texPosition.Y, 0.0f,
WorldView.WORLD_SIZE );

return this.grid.HeightmapAltitude( texPosition );
}
```

```
Grid

//--------------------------------------------------------------------------
public float HeightmapAltitude( Vector2 texturePosition )
{
  int textureIndex = ( ((int)texturePosition.Y) * WorldView.WORLD_SIZE) +
(int)texturePosition.X;

  return (float)(this.heightMapData[textureIndex].A * HEIGHT_MAP_SCALE);
}
```

6.2 Finite State Machine

The code on this tool can be divided in two blocks. The first one corresponds with each character itself and the other part with its finite state machine.

The finite state machine main class (*FiniteStateMachine.cs*) is very simple. All the states of a single finite state machine are store in a Dictionary. The more important method on this class is the Update method which checks if a change of state has occurred:

```
protected Dictionary< eOfficerState, IStateNodeInterface > states;

//--------------------------------------------------------------------------
public void Update( GameTime gameTime )
{
  eOfficerState nextState = this.states[this.currentState].GetNext(
this.game, (float)gameTime.TotalRealTime.TotalSeconds );
  this.currentState       = nextState;
}
```

An example of a specific FSM (Soldier) can be found in the next lines as example for the reader. The aim of these classes is to create the different states filling up the dictionary.

```
//--------------------------------------------------------------------------
public FiniteStateMachine_Soldier( Game game ): base( game )
{
  // Fill up the dictionary
  this.states = new Dictionary< eOfficerState, IStateNodeInterface >();
  this.states.Add( eOfficerState.Aim,  new StateNodeAim() );
  this.states.Add( eOfficerState.Idle, new StateNodeIdle_Soldier() );
  this.states.Add( eOfficerState.Walk, new StateNodeWalk() );
  this.states.Add( eOfficerState.Run,  new StateNodeRun() );
  this.states.Add( eOfficerState.MeleeImpact, new StateImpact_Soldier() );
  this.states.Add( eOfficerState.KnockedOver, new StateKnocked_Soldier());
  this.states.Add( eOfficerState.Die,         new StateNodeDie() );
  this.states.Add( eOfficerState.SwitchWeapon,new StateSwitchWeapon() );
}
```

This makes very easy to add new characters or new behaviours to an existent one due to there is one class per state. The only information that each state has is the input necessary to access to a possible next state, thus the accessible states from that point.

In the next lines the reader can find the Soldier's impact state as example. The most important relevant thing for each state is the fact that depending on the input a new transition will be triggered. It is worth mentioning here that, obviously, not every state can be accessed for the others. As it was explained previously the state flow depends on the finite state machine diagrams.

```
//--------------------------------------------------------------------------
public eOfficerState processInput( Game game, float timeElapsed )
{
  IInputManagerInterface inputManager =
game.Services.GetService(typeof(IInputManagerInterface));
  IFiniteStateMachineService  fsm =
game.Services.GetService(typeof(IFiniteStateMachineService ) );

  // Die
  if( fsm.IsDead() )
  {
    return eOfficerState.Die;
  }
  // Walk
  if( inputManager.GetPushedState( eInput.Walk ) )
  {
    return eOfficerState.Walk;
  }
  // Run
  if( inputManager.GetPushedState( eInput.Run ) )
  {
    return eOfficerState.Run;
  }
  // Knocked over
  if( inputManager.GetPushedState( eInput.KnockedOver ) )
  {
    fsm.ReceiveDamageKnocked();
    return eOfficerState.KnockedOver;
  }
  // Melee Impact
  if( inputManager.GetPushedState( eInput.Impact ) )
  {
    fsm.ReceiveDamageImpact();
    return eOfficerState.MeleeImpact;
  }
  return eOfficerState.Idle;
}
```

Moreover, each state implements a quick way to determine if a certain state is accessible from other. This could be very helpful in a game to check behaviours or reactions. In the Finite State Machine tool it is used to display an updated list of all the state of character, highlighting which of them are active in that certain moment.

```
//-----------------------------------------------------------------------------
public bool IsStateAccessible( eOfficerState state )
{
  if( (state == eOfficerState.Idle)          ||
      (state == eOfficerState.Aim)           ||
      (state == eOfficerState.Walk)          ||
      (state == eOfficerState.Run)           ||
      (state == eOfficerState.MeleeImpact)   ||
      (state == eOfficerState.KnockedOver)   ||
      (state == eOfficerState.Die) )
  {
      return true;
  }
      return false;
}
```

As mentioned, adding new states of an existent character is almost trivial given that there are no dependencies between states or machines.

The character classes contain all the information relative to each character like life, position, rotation or model. This last parameter is the most relevant for this project, so it will be explained in following lines. Each model is a *ModelAsset* object. This class has all the necessary information to import an animated model from SoftImage ModTool(XSi) to XNA through the XSIXNARuntime pipeline and to draw it.

```
public Model  CrosswalkModel; // 3D Model
public int    AnimationIndex    = 0;
public int    OldAnimationIndex = 0;
public List<XSIAnimationContent>    Animations; // Animations container
private XSISASContainer SASData = new XSISASContainer();

//-----------------------------------------------------------------------------
public void LoadContent(String AssetPath, ContentManager content)
{
   this.InitializeLigths();

   CrosswalkModel = content.Load<Model>(AssetPath);
   Animations     = new List<XSIAnimationContent>();

   // Post process animation
   XSIAnimationData l_Animations = CrosswalkModel.Tag as XSIAnimationData;
   if( l_Animations != null )
   {
       foreach( KeyValuePair<String, XSIAnimationContent> AnimationClip in
l_Animations.RuntimeAnimationContentDictionary )
       {
           AnimationClip.Value.BindModelBones( CrosswalkModel );
           Animations.Add( AnimationClip.Value );
       }
       l_Animations.ResolveBones( CrosswalkModel );
    }
}
```

```
//------------------------------------------------------------------------
public void DrawModel( Game game, Camera camera, CharacterModel character )
{
   Model model         = character.GetModel();
   float rotation     = character.GetRotation();
   Vector3 position   = character.GetPosition();
   Matrix worldMatrix = Matrix.CreateRotationY( rotation ) *
Matrix.CreateTranslation( position );

   UpdateSASData( game, camera );
   Matrix[] bones        = GetBones( model );
   bool      isSkinned   = (bones.Length > 0);
   Matrix[] transforms  = new Matrix[model.Bones.Count];

   model.CopyAbsoluteBoneTransformsTo( transforms );
   foreach( ModelMesh mesh in model.Meshes )
   {
      this.SASData.Model = transforms[mesh.ParentBone.Index] * worldMatrix;
      this.SASData.ComputeModel();
      foreach (Effect effect in mesh.Effects)
      {
         SetTheTechnique( isSkinned, effect );

         if( isSkinned )
         {
            BindBones( bones, effect );
         }
         BindOtherParameters( effect );
      }
      mesh.Draw();
  }
}
```

There is a flaw in the xsi_defaultvs.hlsl shader provide by SoftImage tool. In the VSSkinned vertex shader, the weighted position has not been transformed in the world space. This made the 3D model to ignore every transformation (translation and rotation) made on it. The line of code mistaken is:

```
// Skin the vertex position
float4 weightedposition = mul(IN.position, skinTransform);

float4 weightedposition = mul(mul(IN.position, skinTransform), Model);
```

Moreover, there is a second bug or flaw in the XSIXNARuntime default classes. This time the issue appears when trying to animate individually different instances of the same model. Due to the model and animations are loaded only once (even if there are multiple objects using the same models), models are shared. There are two possible workarounds for this problem.

One possible solution is to have one different file for each instance of the model. This is a bad design solution, due to we are replicating the same file but with a different name. Imagine you need to recreate a crowd in a stadium or multiple clouds moving in the sky. Having hundreds of model files replicated is not an acceptable solution. However, this not a very elegant solution could work if the number of instances of the same model is considerable low.

The second and best solution is to solve the actual flaw from a code point of view. The main problem is that every instance of the same model keeps track of the global currentTime of the game since it was loaded, instead of one per instance. Remember that each model is only loaded once. The way to get around this is to:

1. <u>Keep a local TimeSpan per instance of the model</u>. Each object of a model will store its local time span. This time will be updated in the Update method of the current model. This method it is also responsible for the calculations of the locomotion and of checking a possible change of state depending on the input.

As an example, the Soldier Update method is included in the next lines.

```
SoldierModel: CharacterModel

//-----------------------------------------------------------------------
public override void UpdateCurrentModel( Game game, Camera camera,
GameTime gameTime, FiniteStateMachine finiteStateMachine )
{
  IInputManagerInterface inputManager = game.Services.GetService( typeof(
IInputManagerInterface ) );
  TimeSpan elapsedTime=TimeSpan.FromTicks(gameTime.ElapsedGameTime.Ticks);

  CheckForThumbstickValue( inputManager, this.currentAnimation,
eSoldierAnimation.Run );
  if( animationTimeLimits[(int)this.currentAnimation] == 0.0f )
  {
    CheckForLocomotion( inputManager, gameTime, camera, SPEED );
  }
  UpdateAnimationState( elapsedTime.TotalSeconds, finiteStateMachine );
  TimeSpan aimTime = TimeSpan.FromMilliseconds( animationTimeLimits[5]/2);
  if( this.currentAnimation != eSoldierAnimation.Aim )
  {
    Update( gameTime, (int)this.currentAnimation );
  }
  else if( this.animationTimeElapsed.CompareTo(aimTime) < 0 )
  {
    Update( gameTime, (int)this.currentAnimation );
  }
}
```

```
CharacterModel

//--------------------------------------------------------------------------
public void Update( GameTime gameTime, int currentAnimation )
{
  TimeSpan elapsedTime  =
TimeSpan.FromTicks((gameTime.ElapsedGameTime.Ticks * thumbstick));

  this.currentAnimation      = currentAnimation;
  this.gameTime              = gameTime;
  this.animationTimeElapsed += gameTime.ElapsedGameTime;
}
```

2. Create an extension method of PlayBack(). The method provided by the Softimage ModTool runtime pipeline has to be overwritten by an extension method. This extension method uses the old one but taking into account the local currentTime of each instance that has been stored locally in the model class. It also uses a blend variable that is a parameter in the model class stored in order to make blending between different animations when certain change of state occurs.

```
XSIXNARuntimeExtensions
//--------------------------------------------------------------------------
-
public static void PlayBackAt( this XSIAnimationContent Animations,
TimeSpan time, float blend )
{
  if((Animations.Loop) && (Animations.Duration > TimeSpan.Zero))
  {
    long ticks = time.Ticks % Animations.Duration.Ticks;
    time       = TimeSpan.FromTicks(ticks);
  }
  Animations.CurrentTime = time;
  foreach( KeyValuePair<string, XSIAnimationChannel> channel in
Animations.Channels )
  {
    channel.Value.PlayBack(Animations.CurrentTime, blend);
  }
}
```

Please note that this method should be called instead of the original PlayBack().

3. Call the *PlayBackAt* routine before the instance is drawn. The Render function will perform the blending between animations. The order of Update-Render-Draw is important for the success of this technique. The `elapsedTime` takes into account the value of the `thumbstick` (only valid if the input is received by a gamepad controller). This will allow certain animations to play faster or slower depending on how much the thumbstick is pressed. This is especially important for the run animation of the soldier character.

```
CharacterModel

//-----------------------------------------------------------------------
public void DrawModel( Game game, Camera camera )
{
  Render();
  this.model.DrawModel( game, camera, this );
}

//-----------------------------------------------------------------------
private void Render()
{
  TimeSpan  elapsedTime = TimeSpan.FromTicks(
(long)(gameTime.ElapsedGameTime.Ticks * this.thumbstick) );

  if( this.model.Animations.Count > 0)
  {
    if( this.currentBlendTime < this.blendTime )
    {
      this.model.Animations[this.model.OldAnimationIndex].PlayBackAt(
TimeSpan.Parse("0"), 1.0f );
      this.currentBlendTime += (float)elapsedTime.TotalSeconds;
    }
    else
    {
      this.model.OldAnimationIndex  = this.model.AnimationIndex;
      this.currentBlendTime         = this.blendTime;
    }
    if( this.model.AnimationIndex < this.model.Animations.Count )
    {
      if ( this.blendTime != 0.0 )
      {
        Blend = this.currentBlendTime / this.blendTime;
      }
      this.model.Animations[currentAnimation].PlayBackAt(
animationTimeElapsed, Blend );
    }
  }
}
```

A *PerformanceTest* class was implemented to test the capabilities of the finite state machine. The class simulates a crowd displaying 32 random characters in rows and columns. The test demonstrates how to display multiple instances of the same model and tests the power of the tool to animate and displayed a great amount of models at the same time. Each character on the crowd has a list of possible states, avoiding states like Die. In addition to this, a new Finite State Machine class is needed in order to recreate totally random changes of state, independently from the input. The reader can find the functionality explained previously in the following pieces of code.

```
PerformanceTest

//-----------------------------------------------------------------------
public PerformanceTest( Game game )
{
  this.modelPosition.X  = FIRST_COLUMN_X;
  this.modelPosition.Z  = FIRST_ROW_Z;
  this.characters       = new Character[ARRAY_LENGTH];
  int characterIndex    = 0;

  CreateListRandomStates();

  for( int z = 0; z < COLUMN_LENGTH; z++ )
  {
    for( int x = 0; x < ROW_WIDTH; x++ )
    {
      switch( MoreMathHelper.GetRandomInteger( TOTAL_POSSIBILITIES ) )
      {
        case 0:
        case 1:
            FiniteStateMachine_Randomizer fsmSoldier = new
FiniteStateMachine_Randomizer( this.listSoldierRandomStates, game );
            AddCharacter< Character, SoldierModel >( fsmSoldier,
characterIndex++, z );
            break;
        case 2:
        case 3:
            FiniteStateMachine_Randomizer fsmCitizen = new
FiniteStateMachine_Randomizer( this.listCitizenRandomStates, game );
            AddCharacter< Character, CitizenModel >( fsmCitizen,
characterIndex++, z );
            break;
      case 4:
            FiniteStateMachine_Randomizer fsmEchinod = new
FiniteStateMachine_Randomizer( this.listEchinodRandomStates, game );
            AddCharacter< Character, EchinodModel >( fsmEchinod,
characterIndex++, z );
            break;
      default:
            FiniteStateMachine_Randomizer fsmUrchin  = new
FiniteStateMachine_Randomizer( this.listUrchinRandomStates, game );
            AddCharacter< Character, UrchinModel >( fsmUrchin,
characterIndex++, z );
            break;
      }
    }
    this.modelPosition.X = FIRST_COLUMN_X;
  }
}

//-----------------------------------------------------------------------
private void AddCharacter< TCharacter, TCharacterModel >(
FiniteStateMachine fsm, int characterIndex, int zOffset )
      where TCharacter : Character, new()
      where TCharacterModel : CharacterModel, new()
{
      this.characters[characterIndex] = new TCharacter();
      this.characters[characterIndex].Setup( fsm, new TCharacterModel() );
      this.modelPosition.X    += GAP_BETWEEN_COLUMNS;
      this.modelPosition.Z    = FIRST_ROW_Z + (zOffset * GAP_BETWEEN_ROWS);
      this.characters[characterIndex].SetPosition( this.modelPosition );
```

```
}

//-----------------------------------------------------------------------------
private void CreateListRandomStates()
{
  // Soldier
  this.listSoldierRandomStates = new List< eOfficerState >();
  this.listSoldierRandomStates.Add( eOfficerState.Idle );
  this.listSoldierRandomStates.Add( eOfficerState.Walk );
  this.listSoldierRandomStates.Add( eOfficerState.Run );
  this.listSoldierRandomStates.Add( eOfficerState.Aim )
  […]
}
```

The next code corresponds with the new Finite State Machine class mentioned previously. A random time is calculated and decremented every time the Update method is called. Once this value is zero, a change of state will be calculated randomly. The `numberOfStates` variable corresponds with the length of the list of the possible states.

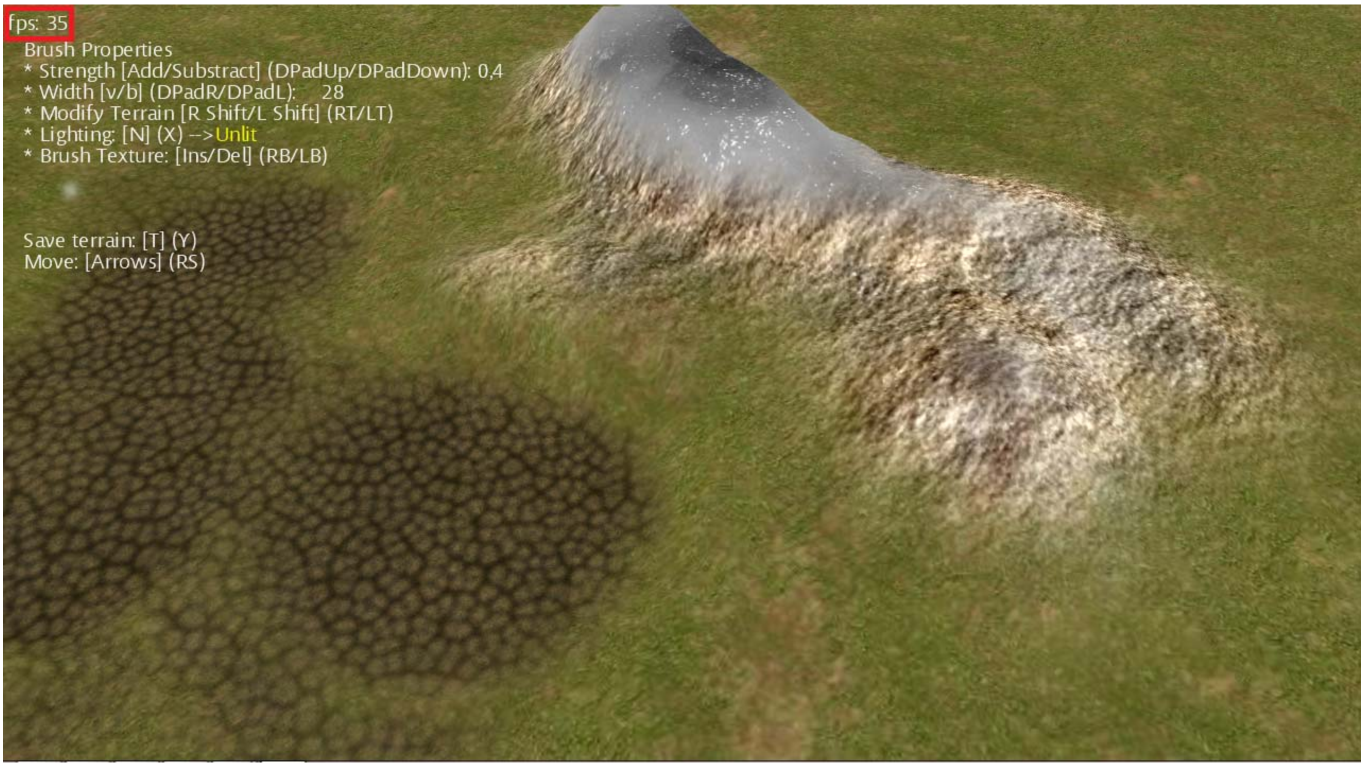
```
FiniteStateMachine_Randomizer : FiniteStateMachine

//-----------------------------------------------------------------------------
public new void Update( GameTime gameTime )
{
  if( this.timeToNextChange <= 0 )
  {
    int newState  = MoreMathHelper.GetRandomInteger( this.numberOfStates );
    this.currentState   = possibleStates[newState];
    // Idle and move states
    if( newState < 2 )
    {
      this.timeToNextChange   = MoreMathHelper.GetRandomFloat(
MIN_MILLISECONDS, MAX_MILLISECONDS );
    }
    else
    {
      this.timeToNextChange   = MIN_MILLISECONDS;
    }
  }
  MathHelper.Clamp(this.timeToNextChange--,0, this.timeToNextChange - 1 );
}
```

7. Demonstration

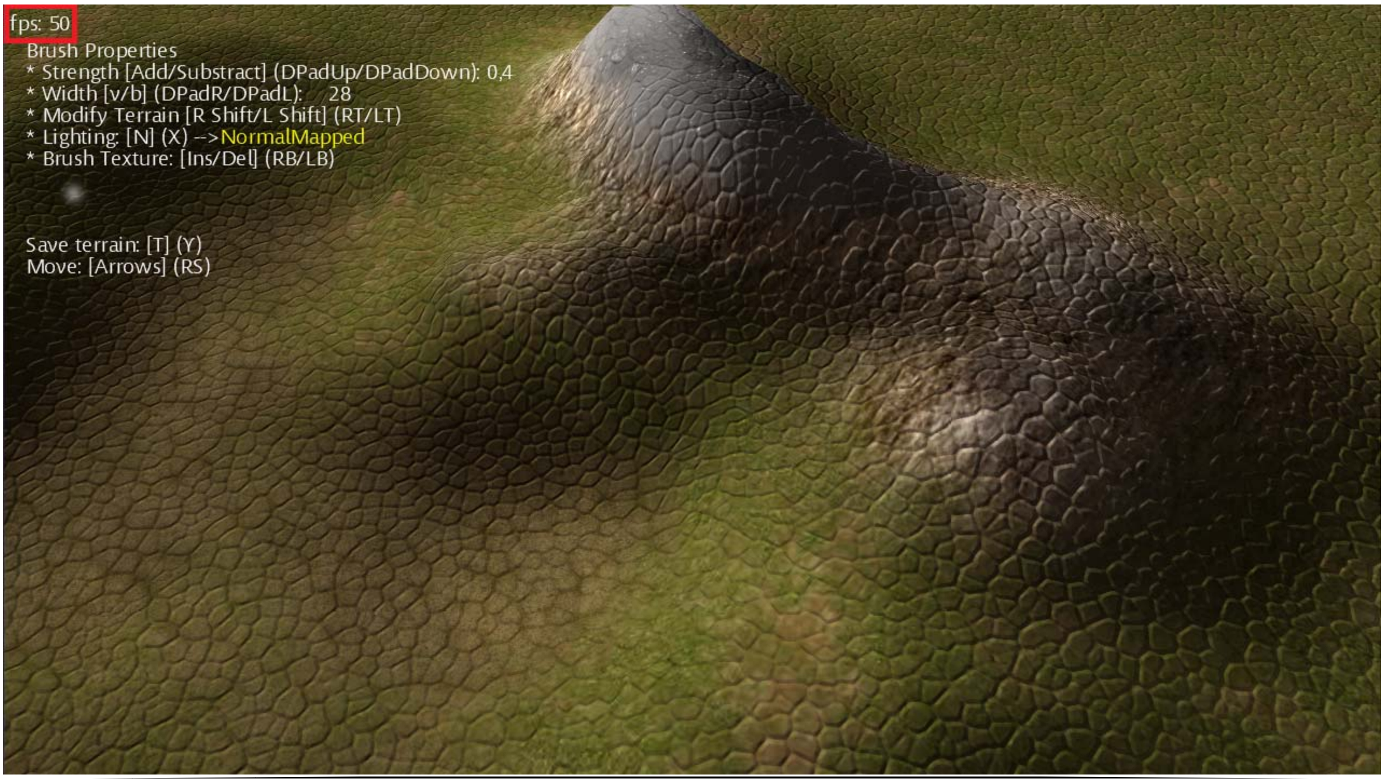
The first set of screenshots was taken directly from the graphic output of the tools. A second set of screenshots from a real example of use of these tools is enclosed.

Concerning the first set of screenshots, the reader will be able to compare the improvement of quality that means the use of the *normal mapping* technique. The following images are screenshots taken from the *Terrain Editor* tool when generating a terrain with the *normal mapping (with CPU and GPU altitude calculations)* able and disable.

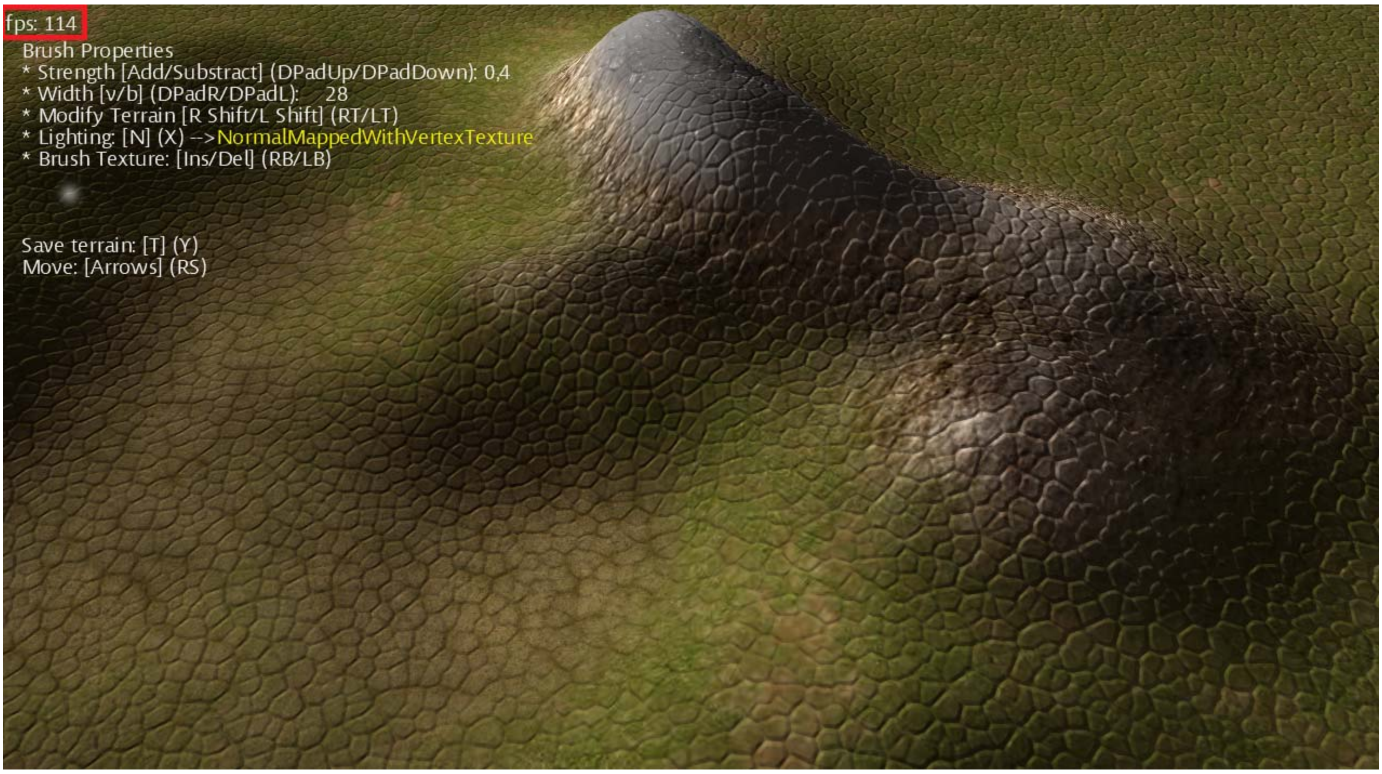
- Unlit: ~35 fps



- Normal mapping: ~50 fps



- Normal mapping with vertex texture: ~115fps



The next set of screenshots corresponds with the FSM tool. Although in this case a video to demonstrate the result of the tool would be more illustrative due to the animation locomotion, the reader will be able to make an idea of the most important features with the following images.

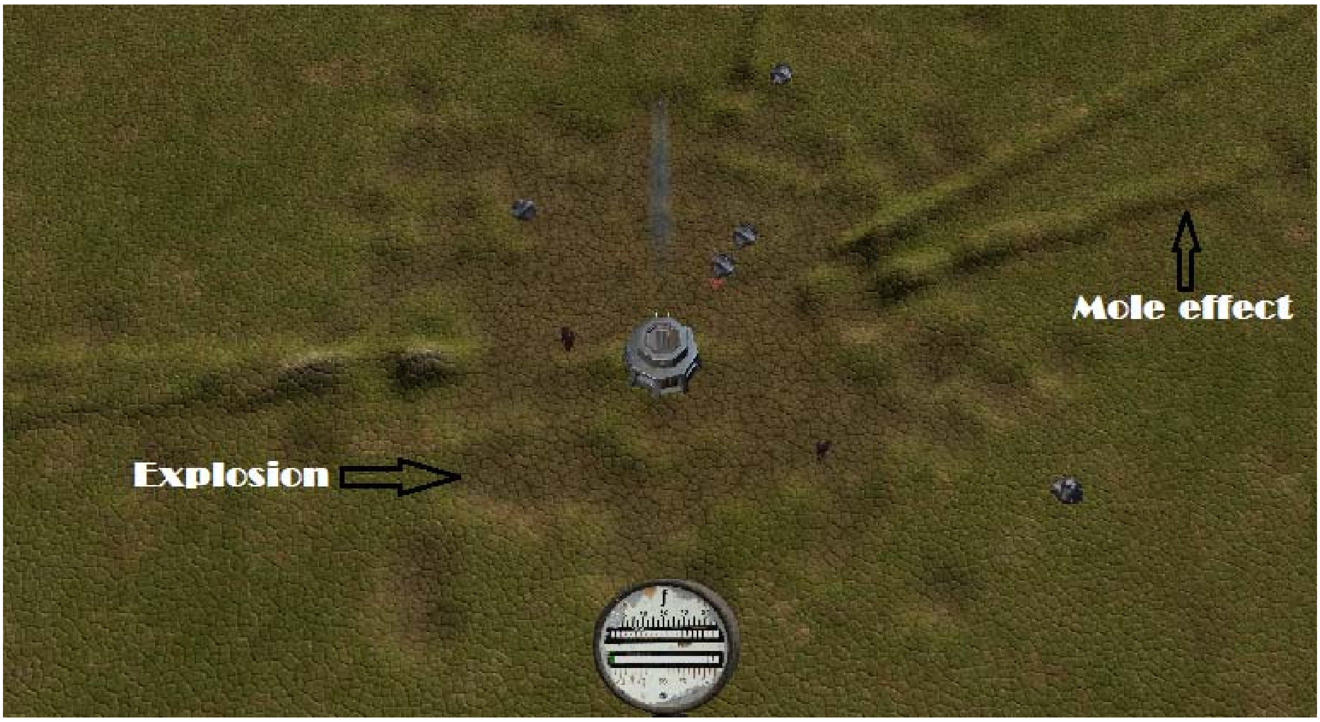
The four different characters implemented:

An example of different character movements depending on the current state:

Performance test:

The last screenshot of this set corresponds with the collision detection of the height map. The reader can see how the characters are placed in a correct position depending on the height of the terrain where they are standing. This is valid for both the main character and the characters in the crowd.

The second set of screenshots corresponds to a complete video game. This video game called *Tremors* has used both tools in its implementation. Tremors is a shooting game for Windows and Xbox360 that thrust players into the role of a soldier in control of a static powerful cannon to defeat increasing waves of different kind of enemies. The first screenshot shows the state of the terrain at the beginning of the game. At this point the surface is completely flat. This is made on purpose to show the differences between the terrain at this point and the terrain when the game finishes. In the second screenshot the terrain has been modified after several waves of enemies.



8. Conclusion

8.1 Evaluation

The XNA framework allows anybody with a basic programming knowledge to develop a simple video game for Windows and Xbox360. Moreover, this is a powerful tool oriented to any type of public, either amateur or professional. This project is in between of those categories. It was developed with amateur tools but designed to be part of a theoretical professional project. Both tools demonstrate what they were designed for; firstly a detailed and high quality modifiable terrain which consumes reasonably less computing process. Secondly, a tool that helps animators to implement their creations into XNA. Thus, it is possible to say that the initial goals have been achieved.

Although the use of a SVN tool has allowed me to track all my progress during the project, I was unable to follow exactly the first Gannt chart designed on October. Nevertheless, this was predictable and therefore the Gannt Chart was redesigned during each iteration. The final result is not far from what was expected on the project proposal document. This means the success of the development process. However, there are some points scheduled in the initial Gannt Chart that have not been accomplished due to time constraints. For example, the attachment of different items and body parts to the characters in the Finite State Machine tool. This will be left to future steps.

8.2 Lessons learned

This project has given me the chance to participate actively in software development cycle, from the design and modelling to the software testing and evaluation.

I consider this project to be very useful for my future. I have gained knowledge not only of techniques used nowadays in video game development (normal and height mapping, Phong model, finite state machines...) but also knowledge about research methodology and finding academic resources within this field. The fact that I have worked in a group of people has given me the chance to improve my communication and team work skills. It also has helped me to write a more readable and understandable code, using industry standards. This work has also helped me to understand the different constraints and difficulties during a software development process like achieving enough constant frame rate, to satisfy all the necessary requirements or making a good design in order to the tool to be easily used.

Initially I planned to implement the Terrain Editor in a first place and the FSM tool afterwards. The order was the best choice due to the complexity of the first tool; however both tools should have overlapped during more time, sharing more resources. Thus, now I'm more aware of the importance of a good scheduling at the beginning of each project. In addition to this, since the Project Proposal I have learned to manage the resources available in order for the development to succeed.

8.3 Future steps

Additional techniques can be added to the tools in order to extend it. In the Terrain Editor tool case, the normal mapping technique can be substituted by *parallax mapping*. This technique, based on the same principals as the one used, is more advanced and offers higher quality results. In addition to this more lighting techniques like shadowing can be implemented in this tool to achieve a more realistic scene. Besides, the Terrain Editor could be extended to be a complete Level Editor by adding the possibility to place characters, objects, buildings and other game elements.

The FSM tool can always be easily extended by adding more characters with their animations. In addition to this, by processing a more accurately blending between each animation the result can look more realistic. It is also possible for the Finite State Machine tool to take advantage of the improvement that means using the GPU for doing complex calculus instead of the CPU. The basic idea is to use a texture to encode all the animations, and then use standard instancing techniques to send both transformation data and an animation frame index. Then look at the texture using vertex texture fetch to pull the animation data out, and finally use that animation data to apply traditional skinned animation. This technique allows a game to have at the same time hundreds or even thousands of animated models achieving a great frame per second count.

Appendix A: references

1.1 References

Chiu, B.; Zordan, V.; Wu, C.(2007) "State-annotated motion graphs". *VRST '07: Proceedings of the 2007 ACM symposium on Virtual reality software and technology*.

Ernst, I.; Rüsseler, H.; Schulz, H.; Wittig, O. (1998) "Gouraud Bump Mapping," Proc. *Eurographics/Siggraph Workshop on Graphics Hardware*, Lisbon, Portugal, pp. 47-53.

Glassner, A.; (1997) Situation normal [Gourand and Phong shading]. Computer Graphics and Applications, IEEE. Volume: 17 , Issue: 2, Page(s): 83 – 87.

Lobão, A.; Evangelista, B.; Leal De Farias, J., Grootjans, R. (2008) "Beginning XNA 3.0 Game Programming: From Novice to Professional". *Apress*.

Muwei J.; Feng G.; Cheng Y.; Junyu D.; Baokang Z.;(2009) Automatic correction of non-uniform illumination for 3D surface heightmap reconstruction. IEEE International Conference on multimedia and Expo. Page(s): 1402 - 1405.

Riemer Grootjans, (2008) "XNA 3.0 Game Programming Recipes: A Problem-Solution Approach". *Apress*.

Sun, J.; Xu, L.; Li, .H; Wu; Q. (2005) "A Practical Bump Mapping Technique in Scene Simulation", Mechatronics and Automation, *IEEE International Conference* Volume 1, pp. 166 - 170.

Zharikova, S.V.; (2002) "Using FSM equations in various applications"

1.2 Image links

[1] http://upload.wikimedia.org/wikipedia/commons/5/57/Heightmap.png
http://upload.wikimedia.org/wikipedia/commons/2/2f/Heightmap_rendered.png

[2] http://digierr.spaces.live.com/blog/cns!2B7007E9EC2AE37B!442.entry

[3] http://upload.wikimedia.org/wikipedia/commons/6/6b/Phong_components_version_4.png

[4] http://www.cs.virginia.edu/~humper/cs150/ps/ps7/phong.gif

[5] http://www.cadtutor.net/dd/bryce/anim/anim.html

Appendix B: resources required

The main tool used by the team is *Microsoft XNA Game Studio*. XNA is a set of tools with a managed runtime environment provided by Microsoft that facilitates computer game development and management. One of the main advantage of it is that is free and available for every person who want to develop a videogame. It is worth to mention at this point that all the software used has this characteristic.

The reader can find in following lines a list of all tools used during the development process:

- *Visual Studio 2008 Express Edition* – Is an Integrated Development Environment (IDE) specially designed to implement XNA code. It includes a debugger and friendly interface.
- *XSI Modtool 6.01* – Is a high-end 3D computer graphics application owned by Autodesk for producing 3D computer graphics, 3D modelling, and computer animation.
- *Xbox360* – The console produced by Microsoft. Both tools can be run and tested on this console.
- *TortoiseSVN* – It is a Subversion client, free released under the GNU General Public License.
- *Beyond Compare* – It is a freeware tool for comparing files and folders. This software is very useful to visualize changes in the code and to integrate different pieces of code in a program.
- *Gantt Project* – is a free cross-platform desktop tool for project scheduling and management.
- *Microsoft Office* – this tool was used to write the reports and documentation of the project.

Appendix C: Tools Guide

1.1 Installing the tool

Each tool has its own installation package. Double click on the execution file (.exe) of the desired tool to install it.

- Terrain Editor tool *code*:

\Terrain Tool**Terrain.sln**

- Terrain Editor *setup* file:

\Terrain Tool\Terrain Executable**Setup.exe**

- Finite State Machine tool *code*:

\FiniteStateMachine Tool**FiniteStateMachine.sln**

- Finite State Machine tool *setup* file:

\FiniteStateMachine Tool\FiniteStateMachine Executable**Setup.exe**

To install a tool: double-click on the desired setup.exe and follow the instructions.
To access the code: open the desired "*.sln* file" with Microsoft Visual C#.

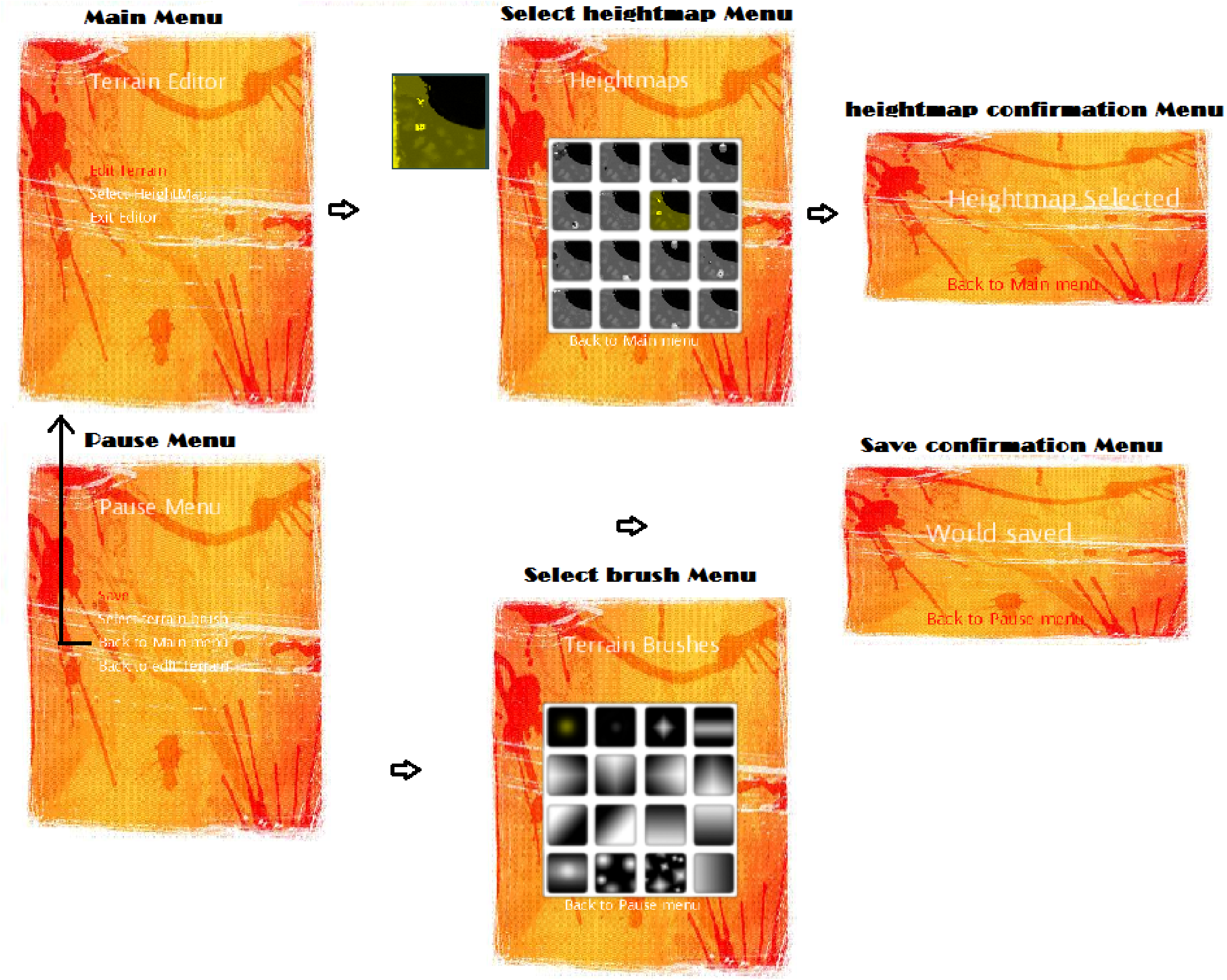
1.2 Terrain Editor controls

To start the tool click go to:

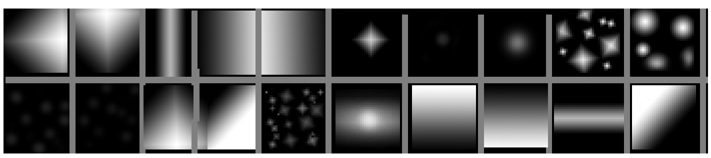
On Windows: Start > Microsoft > Terrain

Once in the Main menu the user can access directly to edit the default map "Edit Terrain" or select "Select HeightMap" to load one of the 16 possible maps.

Terrain Editor main controls		
Movement or action	Keyboard	Gamepad
Normal mapping On/Off	N	X
Raise/Low terrain	Right Shift / Left Shift	RT / LT
Change brush texture	Ins / Del	LB / RB
Change brush strength	Add / Subtract	DPad Up / DPad Down
Change brush size	V / B	DPad Right / DPad Left
Move	Arrows	Left TS
Save	T	Y
Pause Menu	P	Start



The following image shows the 20 different brushes available in the Terrain Tool.



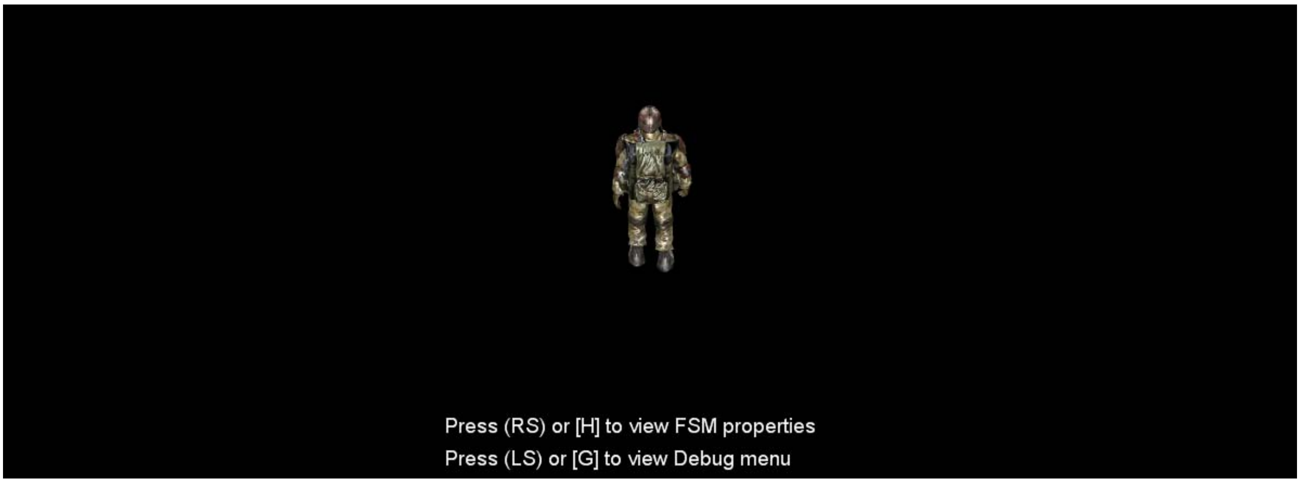
1.3 Finite State Machine controls

To start the tool click go to:

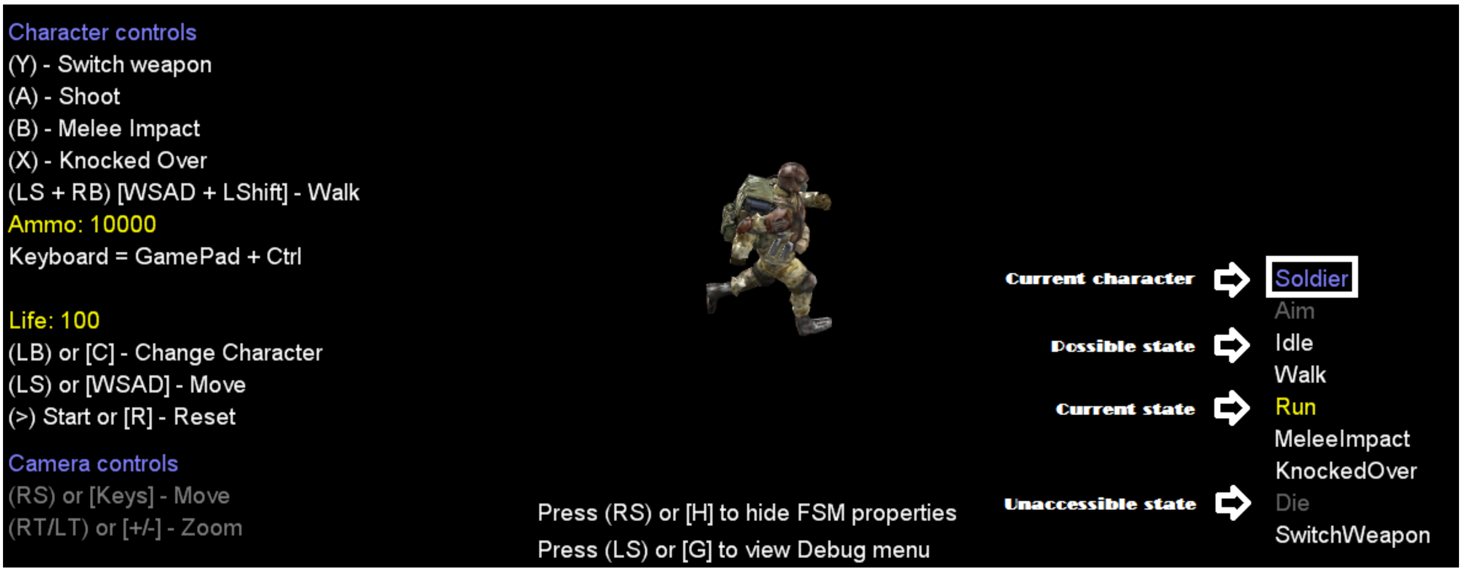
On Windows: Start > Microsoft > Finite State Machine Tool

Automatically the default main character model [Soldier] will be displayed. The tool has two different menus: Finite State Machine Menu and Debug Menu. In the first menu information about the input controls, current character properties and the list of the current finite state machine is displayed. In the second menu the user will find the frame per second counter, the position in the game space of the main character and the input controls to turn on and off the different tests included in the tool. The following screenshots show an explanation of the more relevant elements in each menu:

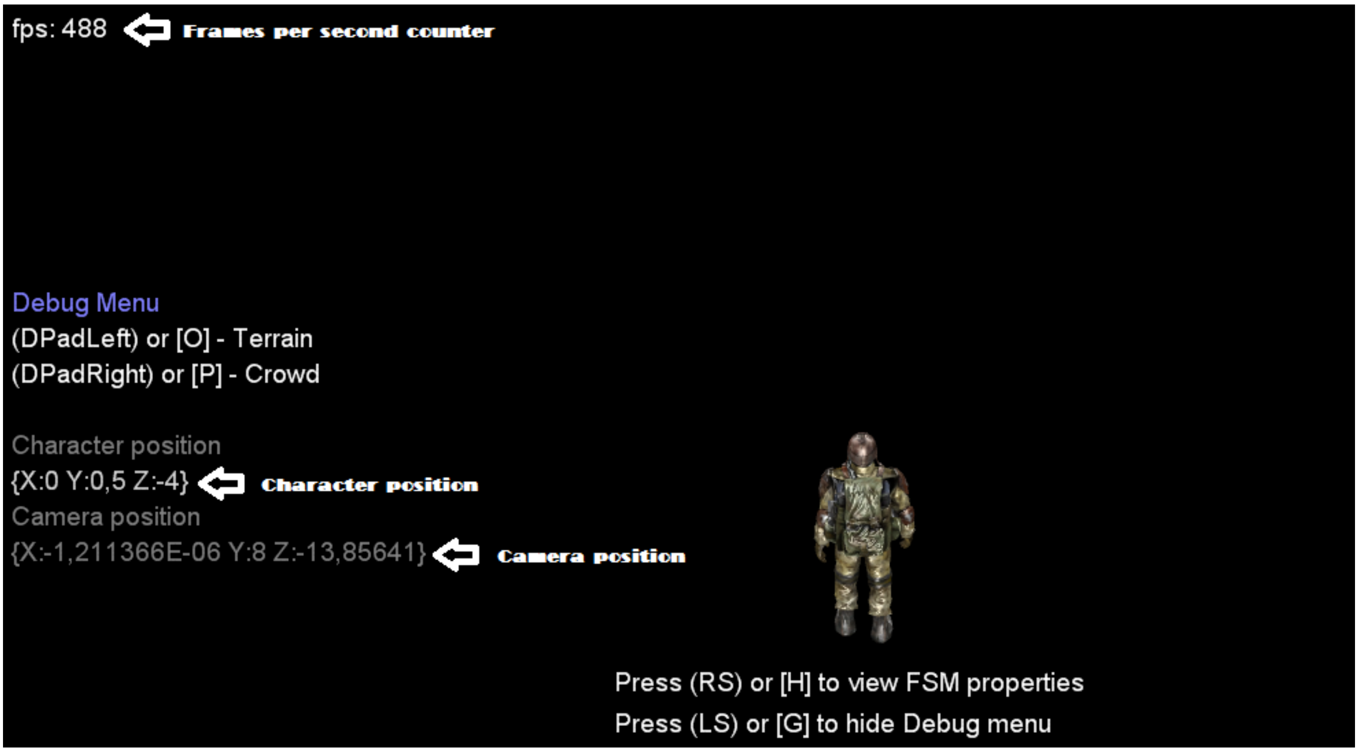
Initial state of the menus:



Finite State Machine Properties menu:



Debug menu:



Finite State Machine tool main controls:

Finite State Machine main controls		
Movement or action	Keyboard	Gamepad
Change character	C	LB
Move	W/A/S/D	Left Stick
Move Camera	Arrows	Right Stick
Zoom	+ / -	RT / LT
Reset Finite State Machine	R	Start
Show/Hide FSM Properties	H	RS
Show/Hide Debug Menu	G	LS
Show/Hide Terrain	O	DPadLeft
Show/Hide Crowd	P	DPadRight

In addition to this, each character has different controls depending on which actions are available for him. The specific controls are displayed on the left part of the screen when the Finite State Machine Menu is displayed. Please note that the Soldier run animation is played at variable speed depending on the left stick input.